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 to implement 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 distributed merge sort, a distributed 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 proved 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 the affine variant [Mostrous and Vasconcelos 2014] of 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 [Oortwijn et al. 2016; Francalanza et al. 2011; Lozes and Villard 2012; Craciun et al. 2015]. These logics typically include some form of mechanism for writing protocol specifications in a high-level manner. Unfortunately, these logics have shortcomings in terms of expressivity. Most importantly, they cannot be used to reason about programs that combine message-passing with other programming and concurrency paradigms, such as higher-order functions, fine-grained shared-memory concurrency, and locks.

In a different line of work, researchers have developed expressive extensions of concurrent separation logic that do support proving strong specifications of programs involving some or all combinations of the aforementioned programming and concurrency paradigms. 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 address message-passing concurrency.

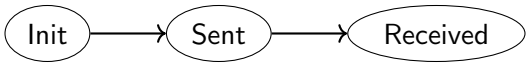
First, there has been work on the use of separation logic to reason about programs that communicate via message passing over a network. The reasoning principles in such logics are geared towards different programming patterns than the ones used in high-level languages like Erlang, Elixir, Go, and Scala. 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]. Second, 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]:

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

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.** Instead of using STSs, Actris extends separation logic with a new 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 binary 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:

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

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 show that $prog_1$ has the specification $\{\mathsf{True}\}\ prog_1\ \{v.\,v = 42\}$, where v is the result of the evaluation.

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 *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:

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

In this program, the main thread sends the number 40 to the forked-off thread, which then adds 2 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'):

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

This protocol states that the second exchanged value is exactly the first with 2 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 $\mathbf{!}\,\langle x\rangle\{\mathsf{True}\}.\,\mathbf{?}\langle x+2\rangle\{\mathsf{True}\}.\,\mathbf{end}$ 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:

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

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:

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

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 $prog_1$:

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

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:

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

This protocol expresses that it is possible to make repeated exchanges with the forked-off thread to increment a number by 2. 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 captures some features of conventional session types. One such example is the *delegation* of channels as seen in the following program:

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

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:

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

The first protocol states that the initial value sent must be a channel endpoint with the protocol used in $prog_1$, meaning that the main thread must give up the ownership of the channel endpoint c_2, thereby delegating it. It 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.

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:

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

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:

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

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 2 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 § 4, 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 capture advanced exchanges, as indicated by the examples in the previous section, they are more restrictive than necessary due to their dual nature. The dual nature of dependent separation protocols 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.

With an asynchronous semantics for message passing, where messages are buffered in both directions, the above notion of duality excludes the verification of certain safe programs. In particular, while it is safe for sends (**!**) to happen before the expected receives (**?**), duality does not allow that. This is demonstrated by the following safe program:

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

Here, both threads first send the value 20, and then 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. With asynchronous message passing, this interaction is safe as neither thread blocks when resolving their send, and both messages can be in transit at the same time because channels are buffered. However, with the features of Actris 1.0 presented in the conference version of this paper [Hinrichsen et al. 2020a], this program cannot be verified as the two dependent separation protocols of channel endpoints must be strictly dual.

Support for type checking such programs has been studied in the session type community, namely in the context of *asynchronous session subtyping* [Mostrous et al. 2009; Mostrous and Yoshida 2015], in which a subtyping relation $st_1 <: st_2$ is defined, capturing that the session type st_2 can be used in place of st_1 when type checking a program. The relation captures that sends can be swapped ahead of receives $\mathbf{?}T.\mathbf{!}U.st <: \mathbf{!}U.\mathbf{?}T.st$. We refer to this as messages being sent *ahead of* the 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** that supports so-called *subprotocols*. Subprotocols are formalised by a preorder $prot_1 \sqsubseteq prot_2$, 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). For example, 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:

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

Since the subprotocol relation is a first-class logical of 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:

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

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. Moreover, 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 foundations for building logical relation models of type systems. In related work by Hinrichsen et al. [2020c], 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 the RustBelt project [Jung et al. 2018a], that work 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 build 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).
- 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 (§ 3).
- 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 (§ 4).
- 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] (§ 5).
- We give a model of dependent separation protocols in the Iris framework to prove safety (*i.e.,* session fidelity) and postcondition validity of our Hoare triples (§ 6).
- We provide a full mechanisation of Actris [Hinrichsen et al. 2020b] 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 (§ 7).

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. 2020a]. 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 additionally extend the model and mechanisation sections 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 section on Actris 2.0's notion of subprotocols (§ 3).
- An updated and expanded description of the model of Actris in Iris (§ 6).
- An expanded section on the Coq mechanisation, with detailed examples of mechanised proofs using the custom tactics for Actris that we have developed (§ 7).

2. A tour of Actris

This section demonstrates the core features of Actris. We first introduce the language (§ 2.1) and the logic (§ 2.2). We then introduce and iteratively extend a simple distributed merge sort algorithm to demonstrate the main features of Actris (§ 2.3–§ 2.8). 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).

2.1. **The Actris language.** The language used throughout the paper is an untyped functional language with higher-order functions, higher-order mutable references, fork-based concurrency, and primitives for message-passing over bidirectional asynchronous channels. The syntax is as follows:

$$v \in \mathsf{Val} ::= () \mid i \mid b \mid \ell \mid c \mid \texttt{rec}\ f\ x = e \mid \ldots \qquad (i \in \mathbb{Z}, b \in \mathbb{B}, \ell \in \mathsf{Loc}, c \in \mathsf{Chan})$$
$$e \in \mathsf{Expr} ::= v \mid x \mid \texttt{rec}\ f\ x = e \mid e_1(e_2) \mid \texttt{ref}\ e \mid !\,e \mid e_1 \leftarrow e_2 \mid$$
$$\texttt{fork}\ \{e\} \mid \texttt{new\_chan}\ () \mid \texttt{send}\ e_1\ e_2 \mid \texttt{recv}\ e \mid \ldots$$

We omit the usual operations on pairs, sums, lists, and integers, which are standard. We introduce the following syntactic sugar: lambda abstractions $\lambda x.\, e$ are defined as $\texttt{rec}\ \_\ x = e$, let-bindings $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$ are defined as $(\lambda x.\, e_2)\ e_1$, and sequencing $e_1; e_2$ is defined as $\texttt{let}\ \_ = e_1\ \texttt{in}\ e_2$. Here, the underscore $\_$ is an anonymous binder, *i.e.,* an arbitrary variable that is fresh in the body of the binding expression.

The language features the usual operations for heap manipulation. New references can be created using $\texttt{ref}\ e$, dereferenced using $!\,e$, and assigned to 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 can be used to implement lock-free data structures and synchronisation primitives, but these are omitted from the syntax.

Message passing is performed over bidirectional channels, which are represented using pairs of buffers $(\vec{v}_1, \vec{v}_2)$ of unbounded size. The $\texttt{new\_chan}\ ()$ 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*. That means, $\texttt{send}\ c_i\ v$ enqueues the value v in its own buffer, *i.e.,* $\vec{v}_i$, and $\texttt{recv}\ c_i$ dequeues a value from the other buffer, *i.e.,* from $\vec{v}_2$ if $i = 1$ and from $\vec{v}_1$ if $i = 2$. Message passing is asynchronous, meaning that $\texttt{send}\ c\ v$ will always reduce, while $\texttt{recv}\ c$ will block as long as the receiving buffer is empty. The exact semantics of the channels will be detailed in § 6.5.

Throughout the paper, we often use the following syntactic sugar to encapsulate the common behaviour of starting a new process:

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

Here, e should evaluate to a function that takes a channel endpoint.

2.2. **The Actris logic.** Actris is a higher-order impredicative concurrent separation logic with a new notion called *dependent separation protocols* to reason about message-passing concurrency. As we will show in § 6, Actris is built as a library on top of the Iris framework [Jung et al. 2015; Krebbers et al. 2017a; Jung et al. 2016, 2018b] and thus inherits all features of Iris. For the purpose of this section, no prior knowledge of Iris is expected as the majority

Grammar:

$$\begin{aligned}
\tau, \sigma ::= {} & x \mid 0 \mid 1 \mid \mathbb{B} \mid \mathbb{N} \mid \mathbb{Z} \mid \mathsf{Type} \mid \forall x : \tau.\, \sigma \mid \\
& \mathsf{Loc} \mid \mathsf{Chan} \mid \mathsf{Val} \mid \mathsf{Expr} \mid \mathsf{iProp} \mid \mathsf{iProto} \mid \ldots \\
t, u, P, Q, prot ::= {} & x \mid \lambda x : \tau.\, t \mid t(u) \mid t(\tau) \mid && \text{(Polymorphic lambda-calculus)} \\
& \mathsf{True} \mid \mathsf{False} \mid P \wedge Q \mid P \vee Q \mid P \Rightarrow Q \mid && \text{(Propositional logic)} \\
& \forall x : \tau.\, P \mid \exists x : \tau.\, P \mid t = u \mid && \text{(Higher-order logic with equality)} \\
& \mu x : \tau.\, t \mid \triangleright P \mid && \text{(Guarded recursion)} \\
& P * Q \mid P \mathrel{-\!\!*} Q \mid \ell \mapsto v \mid \{P\}\, e\, \{v.\, Q\} \mid && \text{(Separation logic)} \\
& c \rightarrowtail prot \mid prot_1 \sqsubseteq prot_2 \mid \overline{prot} \mid prot_1 \cdot prot_2 \mid \mathbf{end} \\
& \mathbf{!}\, \vec{x}{:}\vec{\tau}\, \langle v \rangle \{P\}.\, prot \mid \mathbf{?}\, \vec{x}{:}\vec{\tau}\, \langle v \rangle \{P\}.\, prot \mid \ldots && \text{(Dep. sep. protocols)}
\end{aligned}$$

Ordinary affine separation logic:

AFFINE
$P * Q \Rrightarrow P$

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

HT-VAL
$\{\mathsf{True}\}\, v\, \{w.\, w = v\}$

HT-FORK
$$\frac{\{P\}\, e\, \{\mathsf{True}\}}{\{P\}\, \texttt{fork}\, \{e\}\, \{w.\, w = ()\}}$$

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

Recursion:

HT-REC
$$\frac{\{\triangleright P\}\, e[v/x][\texttt{rec}\ f\ x = e/f]\, \{w.\, Q\}}{\{P\}\, (\texttt{rec}\ f\ x = e)\, v\, \{w.\, Q\}}$$

$\triangleright$-INTRO
$P \Rrightarrow \triangleright P$

LÖB
$(\triangleright P \Rightarrow P) \Rrightarrow P$

μ-UNFOLD
$(\mu x.\, t) = t[\mu x.\, t/x]$

Heap manipulation:

HT-ALLOC
$\{\mathsf{True}\}\, \texttt{ref}\, v\, \{\ell.\, \ell \mapsto v\}$

HT-LOAD
$\{\ell \mapsto v\}\, !\ell\, \{w.\, (w = v) * \ell \mapsto v\}$

HT-STORE
$\{\ell \mapsto v\}\, \ell \leftarrow w\, \{\ell \mapsto w\}$

Message passing:

$$\{\mathsf{True}\}\, \texttt{new\_chan}\, ()\, \{(c, c').\, c \rightarrowtail prot * c' \rightarrowtail \overline{prot}\} \qquad \text{(HT-NEW)}$$

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

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

Dependent separation protocols:

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

Figure 1: The grammar and a selection of rules of Actris.

of Iris's features are orthogonal to Actris's. At this point, we are primarily concerned with Iris's support for nested Hoare triples and guarded recursion, which we need to transfer functions over channels (§ 2.4) and to define recursive protocols (§ 2.6). 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].

The grammar of Actris and a selection of its rules are displayed in Figure 1. The Actris grammar includes the polymorphic lambda-calculus[1] with a number of primitive types and terms operating on these types. Most important is the type iProp of propositions and the type iProto of dependent separation protocols. 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, the meta variable *prot* for protocols, the meta variable v for values, and the meta variables t and u for general terms of any type. Apart from that, there is the implicit side-condition that recursive predicates defined using the recursion operator $\mu x : \tau.\, t$ should be *guarded*. That means, the variable x should appear under a *contractive* term construct. As is usual in logics with guarded recursion [Nakano 2000], the later $\triangleright$ modality is contractive and is used to define recursive predicates. But moreover, as we will demonstrate in § 2.6, the constructors $\mathbf{!}\,\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\,prot$ and $\mathbf{?}\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\,prot$ of dependent separation protocols are contractive in the tail argument *prot* to enable the construction of recursive protocols. The rule μ-UNFOLD says that $\mu x : \tau.\, t$ is in fact a fixpoint of t.

To express program specifications, Actris features Hoare triples $\{P\}\, e\, \{v.\, Q\}$, where P is the precondition and Q the postcondition. The binder v can be used to talk about the return value of e in the postcondition Q, but is omitted if the result is $()$. Note that Hoare triples are propositions of the logic themselves (*i.e.,* they are of type iProp), so they can be nested to express specifications of higher-order functions. The rules for Hoare triples are mostly standard, but it is worth pointing out the rule HT-REC for recursive functions. This rule has a later modality ($\triangleright$) in the precondition, which when combined with the LÖB rule allows reasoning about general recursive functions. As usual, the *points-to* connective $\ell \mapsto v$ expresses unique ownership of a location ℓ with value v. Since we consider a garbage collected language, arbitrary separation logic resources can be discarded via the rule AFFINE.

The novel feature of Actris is its support for dependent separation protocols to reason about message-passing programs. This is done using the $c \rightarrowtail prot$ connective, which expresses unique ownership of a channel endpoint c and states that the endpoint follows the 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. The finite streams are ultimately terminated by an **end** constructor. The value v denotes the message that is being sent (**!**) or received (**?**), the 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 *prot*. For example, $\mathbf{!}\,(b : \mathbb{B})\,(\ell : \mathsf{Loc})\,(i : \mathbb{N})\,\langle (b, \ell)\rangle\{\ell \mapsto i * 10 < i\}.\,prot$ expresses that a pair of a boolean and an integer reference whose value is at least 10 is sent. 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. The $\overline{prot}$ connective denotes the *dual* of a protocol. As with conventional session types, it transforms the protocol by changing all sends (**!**) into receives (**?**),

[1] Actris and Iris, which are both formalised as a shallow embedding in Coq, have in fact a predicative Type hierarchy, while propositions iProp are impredicative. For brevity's sake, we omit details about predicativity of Type, as they are standard.

```
sort_service cmp c ≜
  let l = recv c in
  if |l| ≤ 1 then send c () else
  let l' = split 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;
  merge cmp l l'; send c ()
```

```
sort_client cmp l ≜
  let c = start sort_service cmp in
  send c l;
  recv c
```

Figure 2: A distributed merge sort algorithm (the code for `merge` and `split` is standard and thus elided).

and *vice versa*. Taking the dual twice thus results in the original protocol. The connective $prot_1 \cdot prot_2$ *appends* the protocols $prot_1$ and $prot_2$, which is achieved by substituting any **end** in $prot_1$ with $prot_2$. Finally, $prot_1 \sqsubseteq prot_2$ states that the protocol $prot_1$ is a *subprotocol* of $prot_2$. The subprotocol relation and its proof rules will be described in § 3.

The rule Ht-new allow ascribing any protocol to newly created channels using `new_chan` (), obtaining ownership of $c \rightarrowtail prot$ and $c' \rightarrowtail \overline{prot}$ 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 (*i.e.,* session fidelity, see § 6.7).

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 send to match up with the ascribed value. To send a message w, we need to give up ownership of $c \rightarrowtail \mathbf{!}\,\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\,prot$, 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 $c \rightarrowtail prot[\vec{t}/\vec{x}]$.

The rule Ht-recv for `recv` c is essentially dual to the rule Ht-send. We need to give up ownership of $c \rightarrowtail \mathbf{?}\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\,prot$, 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 $c \rightarrowtail prot[\vec{y}/\vec{x}]$, where $\vec{y}$ are instances of the variables of the protocol.

2.3. **Basic protocols.** In order to show the basic features of dependent separation protocols, we will prove the functional correctness of a simple distributed merge sort algorithm, whose code is shown in Figure 2.

The function `sort_client` takes a comparison function *cmp* and a reference to a linked list l that will be sorted using merge sort. The bulk of the work is done by the `sort_service` function that is parameterised by a channel c over which it receives a reference to the linked list to be sorted. If the list is an empty or singleton list, which is trivially sorted, the function immediately sends back a unit value () to inform the caller that the work has been completed, and terminates. Otherwise, the list is split into two partitions using the `split` function, which updates the list in-place so that ℓ points to the first partition, and returns a reference l' to 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 the `merge` function, which updates the list in-place so that l points to the merged list. Finally, the unit value () is sent back along the original channel c.

$$\texttt{sort\_service}_{func}\ c \triangleq \quad \texttt{let}\ cmp = \texttt{recv}\ c\ \texttt{in}\quad \texttt{sort\_service}\ cmp\ c$$

$$\texttt{sort\_client}_{func}\ cmp\ l \triangleq \quad \texttt{let}\ c = \texttt{start}\ \texttt{sort\_service}_{func}\ \texttt{in}\quad \texttt{send}\ c\ cmp;\ \texttt{send}\ c\ l;\ \texttt{recv}\ c$$

Figure 3: A version of the sort service that receives the comparison function over the channel.

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:

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

Here, R is a decidable total relation on an implicit polymorphic 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 c adheres to the following dependent separation protocol:

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

The protocol describes the interaction of sending a list reference, and then receiving a unit value () once the list is sorted and the reference is updated to point to the sorted list. 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. The specification of the service and the client is as follows:

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

There are two important things to note about these specifications. First, the protocol $\mathsf{sort\_prot}$ is written from the point of view of the client. As such, the precondition for `sort_service` requires that c follows the dual. 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 that in § 2.6 and § 2.7.

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.

2.4. **Transferring functions.** The distributed `sort_service` from the previous section (Figure 2) is parametric on a comparison function. To demonstrate Actris's support for reasoning about functions transferred over channels, we verify the correctness of the program $\texttt{sort\_service}_{func}$ in Figure 3, which receives the comparison function over the channel instead of via a lambda abstraction. To verify this program, we extend the protocol sort_prot from § 2.3 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 captures 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:

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

2.5. **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:

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

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

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.

```
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
  iter (λl'. select c left; send c l'; recv c) l;
  select c right
```

Figure 4: A recursive version of the sort service that can perform multiple jobs in sequence (the code for the function `iter`, which applies a function to each element of the list, is standard and has been elided).

2.6. **Recursive protocols.** We will 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 4. The service $\texttt{sort\_service}_{rec}$ contains a loop in which choice is used to either terminate the service, or to sort an individual list using the distributed merge sort algorithm `sort_service` from § 2.3. The client $\texttt{sort\_client}_{rec}$ uses the service to sort a nested linked list l of linked lists. It performs this job by starting a single instance of the service at 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.

It is important to point out that—as is usual in logics with guarded recursion [Nakano 2000]—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 $\mathbf{!}\,\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\, prot$, which, similarly to $\mathbf{?}\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\, prot$, is contractive in the tail protocol $prot$. The specifications of the service and the client are as follows:

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

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

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.

```
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 = new_list () in
  iter (λl'. select c left;
             let c' = recv c in
             push c' k; send c' l') l
  send c right;
  iter recv k
```

Figure 5: A recursive version of the sort service that uses delegation to perform multiple jobs in parallel (the code for the function `push`, which pushes an element to the head of a list, has been elided).

2.7. **Delegation.** Delegation 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. Due to the impredicativity of dependent separation protocols in Actris, reasoning about programs that make use of 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 ownership of protocols $c \rightarrowtail prot'$ 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 5, which allows multiple sorting jobs to be performed in parallel. To enable parallelism, it delegates a new channel c' to an inner sorting service for each sorting job.

The client $\texttt{sort\_client}_{del}$ once again uses the sorting service to sort a nested linked list l of linked lists. The client starts a connection c to the new service, and for each inner list l', it acquires a delegated channel c', over which it sends a pointer l' to the inner list 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 `iter 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:

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

The specifications of the service and the client are as follows:

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

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

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.

2.8. **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

```
sort_service_fg cmp c ≜
  branch c with
   right ⇒ select c right
  | left  ⇒
    let x1 = recv c in
    branch c with
     right ⇒ select c left; send c x1;
              select c right
    | left  ⇒
      let x2 = recv c in
      let c1 = start sort_service_fg cmp in
      let c2 = start sort_service_fg cmp in
      select c1 left; send c1 x1;
      select c2 left; send c2 x2;
      split_fg c c1 c2; merge_fg cmp c c1 c2
    end
  end

split_fg c c1 c2 ≜
  branch c with
   right ⇒ select c1 right;
            select c2 right
  | left  ⇒
    let x = recv c in
    select c1 left; send c1 x;
    split_fg c c2 c1
  end

merge_fg cmp c c1 c2 ≜
  branch c1 with
   right ⇒ assert false
  | left  ⇒
    let x = recv c1 in
    merge^aux_fg cmp c x c1 c2
  end

merge^aux_fg cmp c x c1 c2 ≜
  branch c2 with
   right ⇒ select c left; send c x1;
            transfer c1 c
  | left  ⇒
    let y = recv c2 in
    if cmp x y then
      select c left; send c x;
      merge^aux_fg cmp c y c2 c1
    else
      select c left; send c y;
      merge^aux_fg cmp c x c1 c2
  end

sort_client_fg cmp l ≜
  let c = start sort_service_fg cmp in
  send_all c l; recv_all c l
```

Figure 6: 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).

sent and received. We demonstrate this feature by considering a fine-grained version of the distributed merge-sort service. This version $\texttt{sort\_service}_{fg}$, as shown in Figure 6, requires the input list to be sent element by element, after which the service sends the sorted list back in the same fashion. We use choice to indicate whether the whole list has been sent (`right`) or another element remains to be sent (`left`).

The structure of $\texttt{sort\_service}_{fg}$ 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, as they are trivially sorted. The inductive case is handled by starting two sub-services at c_1 and c_2 that are sent the two initially received elements, respectively, after which the $\texttt{split}_{fg}$ function is used to receive and forward the remaining values to the sub-services alternatingly. Once the `right` flag is received, the $\texttt{split}_{fg}$ function terminates, and the algorithm moves to the second phase in which the $\texttt{merge}_{fg}$ function merges the stream of values returned by the sub-services and forwards them to the parent service.

The $\texttt{merge}_{fg}$ function initially acquires the first value x from the first sub-service, which it uses in the recursive call as the current largest value. The recursive function $\texttt{merge}_{fg}^{aux}$ recursively requests a value y from the sub-service of which the current largest value was not acquired from. It then compares x and y using the comparison function cmp, and forwards the smallest element. This is repeated until the `right` flag is received from either sub-service, after which the remaining values of the other are forwarded to the parent service using the `transfer` function.

The interface of the client $\texttt{sort\_client}_{fg}$ for this service is similar to the previous ones. It takes a reference to a linked list, which is then sorted. It performs this task by sending the elements of the linked list to the sort service using the `send_all` function (which modifies the list in-place by removing the sent elements), and puts the received values back into the linked list using the `recv_all` function (which also modifies the list in-place). A suitable protocol for proving functional correctness of the fine-grained sorting service is as follows:

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

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

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

The protocol is split into two phases $\mathsf{sort\_prot}_{fg}^{head}$ and $\mathsf{sort\_prot}_{fg}^{tail}$, mimicking the behaviour of the program. The $\mathsf{sort\_prot}_{fg}^{head}$ 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 $\mathsf{sort\_prot}_{fg}^{tail}$ 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.

The top-level specification of the service and client are similar to the specifications of the coarse-grained version of distributed merge sort:

$$\begin{array}{ll}
\{\mathsf{cmp_spec}\ I\ R\ cmp * c \rightarrowtail \overline{\mathsf{sort_prot}_{fg}} \cdot prot\} & \{\mathsf{cmp_spec}\ I\ R\ cmp * \ell \overset{\text{list}}{\mapsto}_I \vec{x}\} \\
\quad \mathsf{sort_prot}_{fg}\ c & \quad \texttt{sort_client}_{fg}\ cmp\ \ell \\
\{c \rightarrowtail prot\} & \{\exists \vec{y}.\ \ell \overset{\text{list}}{\mapsto}_I \vec{y} * \mathsf{sorted_of}_R\ \vec{y}\ \vec{x}\}
\end{array}$$

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.

3. Subprotocols

This section describes **Actris 2.0**, which extends Actris 1.0—as presented in the conference version of this paper [Hinrichsen et al. 2020a]—with *subprotocols*, inspired by asynchronous subtyping of session types. Subprotocols have two key features. First, they expose the

asynchronous nature of channels in the Actris logic by relaxing the requirements 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 and not equality, thereby providing more flexibility in the design and reuse of protocols.

We first introduce the subprotocol relation and its proof rules (§ 3.1). 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 (§ 3.2). Next, we verify a list reversal service whose protocol involves a minimal specification of lists, which we then reuse through subprotocols to obtain a protocol with a more generic specification of lists (§ 3.3). Finally, we show that the subprotocol relation is coinductive, and thereby when combined with Löb induction, can be used to reason about recursive procotols (§ 3.4).

3.1. **The subprotocol relation.** The dependent separation protocol of a channel is picked upon channel creation (using the rule HT-NEW), 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, strictly working with a channel's protocol and its dual is more restrictive than necessary, as either endpoint is agnostic to some variance from the protocol by the other endpoint. We capture the flexibility of safe variances via a new notion—the *subprotocol relation*:

$$prot_1 \sqsubseteq prot_2$$

This 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 the 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 are sound, as (1) the originally expected type that is to be sent can be derived from the subtype, (2) the originally expected type that is to be received can be derived from the supertype, and (3) sends do not block because channels are buffered in both directions. These variances, along with the swapping property, can be generalised to dependent separation protocols using the following proof rules:

$$\sqsubseteq\text{-SEND-MONO'}\quad \frac{\forall \vec{x}{:}\vec{\tau}.\, P_2 \mathrel{-\!\!*} P_1 \qquad \forall \vec{x}{:}\vec{\tau}.\, prot_1 \sqsubseteq prot_2}{\mathbf{!}\,\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P_1\}.\, prot_1 \sqsubseteq \mathbf{!}\,\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P_2\}.\, prot_2}$$

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

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

The rules $\sqsubseteq$-SEND-MONO' and $\sqsubseteq$-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 $\sqsubseteq$-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.

With this set of rules we can make subprotocol derivations such as the following:

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

Here, we strengthen the proposition of the send (by increasing the bound from $j > 42$ to $j > 50$), weaken the proposition of the receive (by reducing the bound from $i < 42$ to $i < 40$), while also swapping the send ahead of the receive.

While the aforementioned rules cover the intuition behind Actris's subprotocol relation, Actris's actual rules for subprotocols 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 key idea to obtain these features is to consider the subprotocol relation $\mathit{prot}_1 \sqsubseteq \mathit{prot}_2$ as a first-class logical connective. That is, $\mathit{prot}_1 \sqsubseteq \mathit{prot}_2$ is an Iris proposition that can be combined freely with the logical connectives of Iris and Actris (*e.g.,* separating conjunction, separating implication, higher-order quantification). Since $\mathit{prot}_1 \sqsubseteq \mathit{prot}_2$ is an Iris proposition, the proof rules are in fact (separating) implications in Iris. For readability, we use inference rules to denote each rule $(P_1 * \cdots * P_n) \mathrel{-\!\!*} Q$ as:

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

The full set of rules for subprotocols is shown in Figure 7. The first four rules account for logical variable manipulation and resource transfer: Rules $\sqsubseteq$-SEND-OUT and $\sqsubseteq$-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 $\sqsubseteq$-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 $\sqsubseteq$-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 accordingly:

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

We first use rule $\sqsubseteq$-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 $\sqsubseteq$-SEND-IN

Logical variable manipulation and resource transfer:

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

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

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

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

Monotonicity and swapping:

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

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

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

Reflexivity and transitivity:

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

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

Dual and append:

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

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

Channel ownership:

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

Figure 7: The rules of Actris 2.0 for subprotocols.

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 ($\sqsubseteq$-SEND-MONO and $\sqsubseteq$-RECV-MONO) and swapping ($\sqsubseteq$-SWAP) in Figure 7 differ in two aspects from the rules for monotonicity ($\sqsubseteq$-SEND-MONO' and $\sqsubseteq$-RECV-MONO') and swapping ($\sqsubseteq$-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 $\vec{x}{:}\vec{\tau}$ 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 (§ 3.4).

The remaining rules in Figure 7 express that the subprotocol relation is reflexive ($\sqsubseteq$-REFL) and transitive ($\sqsubseteq$-TRANS), as well as that the dual operation is anti-monotone ($\sqsubseteq$-DUAL) and the append operation is monotone ($\sqsubseteq$-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:

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

Here we extend the protocol $\mathbf{!}\,\langle v\rangle\{P\}.\,\mathbf{?}\langle w\rangle\{Q\}.\,\mathit{prot}$ with a so-called *frame* R, which describes resources that must be sent along with the originally expected resources P, and which are reacquired along with the resources Q that are sent back. 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:

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

We use rule $\sqsubseteq$-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 $\sqsubseteq$-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 $\sqsubseteq$-SEND-MONO to proceed with the receiving part of the protocol in a dual fashion—we use rule $\sqsubseteq$-RECV-OUT to transfer out Q of the stronger protocol (*i.e.,* the receive on the LHS), and use rule $\sqsubseteq$-RECV-IN to transfer Q and the frame R into the weaker protocol (*i.e.,* the receive on the RHS).

3.2. **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 8. The service `mapper_service` is a loop, which iteratively receives an element, maps a function over that element, and sends the resulting value back. Conversely, the client `mapper_client` 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 former interleaves the sends and receives, while the latter does not, the dependent

```
mapper_service f_v c ≜
  branch c with
   left  ⇒ send (f_v (recv c)) ;
            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 8: A mapper service whose verification relies on swapping (the code for the functions `send_all` and `recvN` has been elided).

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 interleaved communication:

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

The protocol is parameterised by representation predicates I_T and I_U that relate language-level 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 language-level function f_v. The connection between f and f_v is formalised as:

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

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 from § 2. However, to verify `mapper_client` against the protocol `mapper_prot`, we need to weaken the protocol using Actris's rules for subprotocols. Given a list of n elements, the subprotocol relation (together with an intermediate step) that describes this weakening is:

$$\begin{array}{ll}
& \texttt{mapper_prot}\ I_T\ I_U\ f \\
\sqsubseteq & \mathbf{!}\,\langle\texttt{left}\rangle.\,\mathbf{!}\,(x_1 : T)\ (v_1 : \mathsf{Val})\,\langle v_1\rangle\{I_T\ x_1\ v_1\}. \\
& \mathbf{?}(y_1 : U)\,\langle y_1\rangle\{I_U\ (f\ x_1)\ y_1\}.\ \cdots \\
& \mathbf{!}\,\langle\texttt{left}\rangle.\,\mathbf{!}\,(x_n : T)\ (v_n : \mathsf{Val})\,\langle v_n\rangle\{I_T\ x_n\ v_n\}. \\
& \mathbf{?}(y_n : U)\,\langle y_n\rangle\{I_U\ (f\ x_n)\ y_n\}. \\
& \texttt{mapper_prot}\ I_T\ I_U\ f \\
\sqsubseteq & \mathbf{!}\,\langle\texttt{left}\rangle.\,\mathbf{!}\,(x_1 : T)\ (v_1 : \mathsf{Val})\,\langle v_1\rangle\{I_T\ x_1\ v_1\}.\ \cdots \\
& \mathbf{!}\,\langle\texttt{left}\rangle.\,\mathbf{!}\,(x_n : T)\ (v_n : \mathsf{Val})\,\langle v_n\rangle\{I_T\ x_n\ v_n\}. \\
& \mathbf{?}(y_1 : U)\,\langle y_1\rangle\{I_U\ (f\ x_1)\ y_1\}.\ \cdots \\
& \mathbf{?}(y_n : U)\,\langle y_n\rangle\{I_U\ (f\ x_n)\ y_n\}. \\
& \texttt{mapper_prot}\ I_T\ I_U\ f
\end{array}$$

n times μ-UNFOLD and weaken $\oplus$ into $\mathbf{!}\,\langle\texttt{left}\rangle$

n times $\sqsubseteq$-SWAP'

Both steps are proven by induction on n. In the first step, we unfold the recursive protocol n times using μ-UNFOLD and the derived rule $(prot_1 \oplus prot_2) \sqsubseteq \mathbf{!}\,\langle\texttt{left}\rangle.\,prot_1$ to weaken the choices. Recall from § 2.5 that $\oplus$ is defined in terms of the send protocol ($\mathbf{!}$), allowing us to prove $(prot_1 \oplus prot_2) \sqsubseteq \mathbf{!}\,\langle\texttt{left}\rangle.\,prot_1$ using $\sqsubseteq$-SEND-OUT and $\sqsubseteq$-SEND-IN. The second step involves swapping all sends ahead of the receives using the rule $\sqsubseteq$-SWAP'.

```
list_rev_service c ≜                    list_rev_client ℓ ≜
  let ℓ = recv c in list_rev ℓ; send ()   let c = start list_rev_service in
                                          send c ℓ; recv c
```

Figure 9: A list reversing service (the code for the function `list_rev` has been elided).

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 prove the following specifications for the service and the client:

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

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

3.3. **Minimal protocols.** An essential feature of separation logic is the ability to assign strong and minimal specifications to libraries, so that each library can be verified once against its specification, which in turn can be used to verify as many client programs as possible. To achieve a similar goal for message-passing programs we would like to assign strong and minimal protocols to services, so that each service can be verified once against its protocol, which in turn can be used to verify as many clients as possible. One of the key ingredients of separation logic to allow for such specifications is the frame rule (HT-FRAME). In § 3.1 we showed that subprotocols allow framing in protocols. In this section we give a detailed example of framing in protocols by considering the service `list_rev_service` in Figure 9, which receives a linked list, reverses it, and sends it back.

To specify this service, we could use a protocol similar to the sorting service in § 2.3, 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 $\overline{\mathsf{list\_rev\_prot}_{I_T}}$, that 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 minimal 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:

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

Once we have verified the service against the minimal protocol, a client might still want to interact with the list reversal service through the protocol $\mathsf{list\_rev\_prot}_{I_T}$. This can be achieved by proving the subprotocol relation $\mathsf{list\_rev\_prot} \sqsubseteq \mathsf{list\_rev\_prot}_{I_T}$. To do so, we first establish a relation between the two versions of the list representation predicate:

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

Here, $\ast_{(x,v)\in(\vec{x},\vec{v})}$ is the pair-wise iterated separation conjuction 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:

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

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 § 3.1, 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:

$$\begin{array}{ll}
\{c \rightarrowtail \mathsf{list_rev_prot} \cdot \mathit{prot}\} & \big\{\ell \overset{\text{list}}{\mapsto}_{I_T} \vec{x}\big\}\\
\quad \texttt{list_rev_service}\ c & \quad \texttt{list_rev_client}\ \ell\\
\{c \rightarrowtail \mathit{prot}\} & \big\{\ell \overset{\text{list}}{\mapsto}_{I_T} \mathsf{reverse}\ \vec{x}\big\}
\end{array}$$

3.4. **Subprotocols and recursion.** We conclude this section by showing how subprotocol relations involving recursive protocols can be proved using LÖB induction. Recall from § 2 that the principle of LÖB induction is as follows:

$$(\triangleright P \Rightarrow P) \Rightarrow P$$

By letting P to be $\mathit{prot}_1 \sqsubseteq \mathit{prot}_2$, this means that we can prove $\mathit{prot}_1 \sqsubseteq \mathit{prot}_2$ under the assumption of the induction hypothesis $\triangleright(\mathit{prot}_1 \sqsubseteq \mathit{prot}_2)$. The later modality ($\triangleright$) ensures that we do not immediately use the induction hypothesis, but first apply the monotonicity rule for send ($\sqsubseteq$-SEND-MONO) or receive ($\sqsubseteq$-RECV-MONO), which 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 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:

$$\begin{aligned}
\mathit{prot}_1 &\triangleq \mu(\mathit{rec} : \mathsf{iProto}).\,(\mathsf{list_rev_prot} \cdot \mathit{rec}) \oplus \mathbf{end}\\
\mathit{prot}_2 &\triangleq \mu(\mathit{rec} : \mathsf{iProto}).\,(\mathsf{list_rev_prot}_{I_T} \cdot \mathit{rec}) \oplus \mathbf{end}
\end{aligned}$$

Here, $\mathsf{list\_rev\_prot}$ and $\mathsf{list\_rev\_prot}_{I_T}$ are the protocols from § 3.3, for which we already proved $\mathsf{list\_rev\_prot} \sqsubseteq \mathsf{list\_rev\_prot}_{I_T}$. The proof is as follows:

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

The proof starts with rule LÖB and unfolding the recursive types. We then proceed with monotonicity of $\oplus$, *i.e.,* $\triangleright(prot_1 \sqsubseteq prot_2 \land prot_3 \sqsubseteq prot_4) \mathrel{-\!\!*} (prot_1 \oplus prot_3) \sqsubseteq (prot_2 \oplus prot_4)$, which itself follows from $\sqsubseteq$-SEND-MONO since selection ($\oplus$) is defined in terms of send (**!**). In step $(*)$, we use $\mathsf{list\_rev\_prot} \sqsubseteq \mathsf{list\_rev\_prot}_{I_T}$, which we proved in § 3.3.

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:

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

After we use $\sqsubseteq$-SEND-MONO' for the first time, we strip off the later of the induction hypothesis $\triangleright(prot_1 \sqsubseteq prot_2)$. Subsequently, when we use $\sqsubseteq$-SEND-MONO' and $\sqsubseteq$-RECV-MONO, there are no more laters to strip. We therefore introduce the later using $\triangleright$-INTRO before applying the appropriate monotonicity rule.

$$\{R\}\ \texttt{new\_lock}\ ()\ \{lk.\ \mathsf{is\_lock}\ lk\ R\} \qquad \text{(HT-NEW-LOCK)}$$
$$\{\mathsf{is\_lock}\ lk\ R\}\ \texttt{acquire}\ lk\ \{R\} \qquad \text{(HT-ACQUIRE)}$$
$$\{\mathsf{is\_lock}\ lk\ R * R\}\ \texttt{release}\ lk\ \{\mathsf{True}\} \qquad \text{(HT-RELEASE)}$$
$$\mathsf{is\_lock}\ lk\ R \mathrel{-\!\!*} \mathsf{is\_lock}\ lk\ R * \mathsf{is\_lock}\ lk\ R \qquad \text{(LOCK-DUP)}$$

Figure 10: The rules Actris inherits from Iris for locks.

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

Figure 11: A sample program that combines locks and channels to achieve manifest sharing.

4. 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 locks and ghost state that Actris readily inherits from Iris. We present the basic idea with a simple introductory example of sharing a channel endpoint between two parties (§ 4.1). We then consider a more challenging example of a distributed load-balancing mapper (§ 4.2).

4.1. Locks and ghost state.

Using the language from §2.1 it is possible to implement locks using a spin lock, ticket lock, or a more sophisticated implementation. For the purpose of this paper, we abstract over the concrete implementation and assume that we have operations `new_lock`, `acquire` and `release` that satisfy the common separation logic specifications for locks as shown in Figure 10.

The `new_lock` () operation creates a new lock, which can be thought of as a mutex. The operation `acquire` lk will atomically take the lock or block in the case the lock is already taken, and `release` lk releases the lock so that it may be acquired by other threads. The specifications in Figure 10 make use of the representation predicate $\mathsf{is\_lock}\ lk\ R$, 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 $\mathsf{is\_lock}\ lk\ R$ (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

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

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

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

$$\mathsf{auth}_\gamma\, n * \mathsf{contrib}_\gamma \mathrel{-\!\!*} n > 0 \qquad \text{(Auth-contrib-pos)}$$

Figure 12: The authoritative contribution ghost theory.

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.

To show how locks can be used, consider the program in Figure 11, which uses a lock to share a channel endpoint between two threads, that each send 21 to the main thread. The following dependent protocol, where n denotes the number of messages that should be exchanged, captures the expected interaction from the point of view of the main thread:

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

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 over time depending on the values that are sent. To tie the number n to the number of contributions made by the threads that share the channel endpoint, we make use of the connectives $\mathsf{auth}_\gamma\, n$ and $\mathsf{contrib}_\gamma$, which are defined using Iris's "ghost theory" mechanism for "user-defined" ghost state [Jung et al. 2015, 2018b].

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. These concepts are made precise by the rules in Figure 12. The rule Auth-init expresses that an authority $\mathsf{auth}_\gamma\, 0$ can always be created, which is given some fresh ghost identifier γ. Using the rules Auth-alloc and Auth-dealloc, one can allocate and deallocate tokens $\mathsf{contrib}_\gamma$ as long as the count 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.

Most of the rules in Figure 12 involve the logical connective $\Rrightarrow$ of a so-called *view shift*. The view shift connective, which Actris inherits from Iris, can be though of as a "ghost update", which is made precise by the structural rules Vs-csq and Vs-frame rules, that establish the connection between $\Rrightarrow$ and the Hoare triples of the logic:

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

With the ghost state in place, we can now state 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:

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

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.

4.2. **A distributed 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 distributed load-balancing mapper that maps a function f_v over a list. Our distributed 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, which can be found in Figure 13, 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.[2]

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

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

Similarly to `mapper_prot` from § 3.2, the protocol is parameterised by representation predicates I_T and I_U, and a function $f : T \to \mathsf{List}\ U$ in Iris/Actris logic, related through the `f_spec` specification. Similar to the protocol `lock_prot` from § 4.1, the protocol par_mapper_prot

[2]The entire code is present in the accompanied Coq development [Hinrichsen et al. 2020b].

```
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 13: A worker of the distributed 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 14: The authoritative contribution ghost theory extended with multisets.

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 $(n = 1) \Rightarrow (X = \emptyset)$ 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 § 4.1. 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 14. 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.

The 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 are as follows:

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

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

The lock invariant and specification of `par_mapper_worker` are similar to those used in the simple example in § 4.1. The specification of `par_mapper_client` $n\ f_\mathsf{v}\ \ell$ simply states that the resulting linked list points to 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 $\mathsf{par\_mapper\_prot}$, 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.

5. 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 distributed 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 distributes 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 distributed merge sort algorithm (§ 2.8) and the distributed load-balancing mapper (§ 4.2). 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. 2020b].

5.1. 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 again 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:

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

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

5.2. **Implementation of map-reduce.** The general distributed model of map-reduce is achieved by distributing 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 distributed 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 the mapper nodes, that each apply the "map" function f to their given data in parallel.
(2) Collect the complete set of mapped results and "shuffle" them, *i.e.,* group them by key. The grouping is commonly implemented using a distributed sorting algorithm.
(3) Split the shuffled data into chunks and delegate these chunks to the reducer nodes that each apply the "reduce" function g to their given data in parallel.
(4) Collect and aggregate the complete set of result of the reducers.

Our variant of the map-reduce model is defined as a function $\texttt{map\_reduce}_{\mathsf{v}}\ n\ m\ f_{\mathsf{v}}\ g_{\mathsf{v}}\ \ell$, which coordinates the work for performing map-reduce on a linked list ℓ between n mappers performing the "map" function f_{v} and m workers performing the "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 § 4, parameterised with the f_{v} function. Additionally start an instance of $\texttt{sort\_service}_{fg}$ from § 2, 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 $\mathbb{Z}$ 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.
(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.

5.3. **Functional correctness of map-reduce.** The specification of the map-reduce program is as follows:

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

The $\mathsf{f\_spec}$ predicates (as introduced in § 3.2) establish a connection between the functions f and g on the logical level and the functions f_v and g_v in the language. 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 § 5.1.

6. The model of Actris

We prove the adequacy theorem of Actris—given a Hoare triple $\{\mathsf{True}\}\ e\ \{\phi\}$ that is derivable in Actris, we prove that e cannot get stuck (*i.e.,* safety), and if e terminates, the resulting value v satisfies ϕ (*i.e.,* postcondition validity). To do that, we construct a model of Actris as a shallow embedding in the Iris framework [Jung et al. 2015; Krebbers et al. 2017a; Jung et al. 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. Adequacy of Actris is then simply a consequence of Iris's adequacy theorem.

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 operators for dual and composition are defined (§ 6.1). We then define the subprotocol relation $prot_1 \sqsubseteq prot_2$ and prove its proof rules as lemmas (§ 6.2). 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 (§ 6.3). On top of the protocol consistency relation, we define the *Actris ghost theory* for dependent separation protocols (§ 6.4), which forms the key ingredient for defining the connective $c \rightarrowtail prot$ for channel ownership (§ 6.6) that links protocols to the semantics of channels (§ 6.5). We then show how adequacy follows from the embedding in Iris (§ 6.7). Finally, we show how to solve the recursive domain equation for the type iProto of dependent separation protocols (§ 6.8).

6.1. **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) instead of meta-level predicates (functions to Prop). 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 occurence 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 § 6.8. 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 ($\mu x.\,t$):

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

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

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 1 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. 2020a], 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 § 6.8 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 procotols like $\mu \mathit{rec}.\, !\,(c : \mathsf{Chan})\,\langle c\rangle\{c \rightarrowtail \mathit{prot}\}.\,\mathit{prot}'$, 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.

6.2. **The model of the subprotocol relation.** We now model the subprotocol relation $\mathit{prot}_1 \sqsubseteq \mathit{prot}_2$ from § 3. 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 later modality ($\triangleright$). Aside from the 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 7 as lemmas, and for Actris users to reason about recursive protocols as shown in § 3.4. 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 $!\,\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\,prot$ and $?\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\,prot$, 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:

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

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

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

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 $!\,\langle v_2\rangle.\,prot$ and $?\langle v_1\rangle.\,prot$, 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 7 on page 21 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.

6.3. **Protocol consistency.** To connect dependent separation protocols to the semantics of channels in § 6.6, 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:

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

Intuitively, $\mathsf{prot\_consistent}\ \vec{v}_1\ \vec{v}_2\ prot_1\ prot_2$ ensures that for all messages $\vec{v}_1$ 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), which additionally captures ownership of the resources associated with the messages $\vec{v}_1$ and $\vec{v}_2$.

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

Figure 15: Higher-order ghost variables in Iris.

Closure under the subprotocol relation gives us 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 enqueing 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 proved by inversion on the definition of the subprotocol relation ($\sqsubseteq$). Since the subprotocol relation ($\sqsubseteq$) is defined using guarded recursion, we obtain a later modality ($\triangleright$) for each inversion. To prove the first rule, we need to perform a number of inversions equal to the size of the buffer $\vec{v}_2$, whereas for the second rule we need to perform just a single inversion. The later modalities will be eliminated through physical program steps in the semantics of channels in § 6.5.

6.4. **The Actris ghost theory.** To provide a general interface for adopting Actris's reasoning principles for arbitrary message-passing languages, we employ a standard ghost theory approach to compartmentalise channel ownership.

We use an approach similar to the ghost theory for contributions that we used in § 4. The authority $\mathsf{prot\_ctx}\ (\gamma_1, \gamma_2)\ \vec{v}_1\ \vec{v}_2$ governs the global state of the buffers $\vec{v}_1$ and $\vec{v}_2$. The tokens $\mathsf{prot\_own_l}\ (\gamma_1, \gamma_2)\ prot_l$ and $\mathsf{prot\_own_r}\ (\gamma_1, \gamma_2)\ prot_r$ provide local views of that state, being that the protocols $prot_l$ and $prot_r$ are consistent with the buffers. As we will see in § 6.6, the authority can then be shared through a lock, while the tokens can be distributed to individual threads. The ghost connectives are identified by the shared ghost identifiers γ_1 and γ_2 for the protocols $prot_l$ and $prot_r$, respectively.

To define the connectives of the Actris ghost theory we use Iris's higher-order ghost variables, whose rules are shown in Figure 15. Higher-order ghost variables come in pairs $\gamma \mapsto_\bullet prot$ and $\gamma \mapsto_\circ prot$, which always hold the same protocol $prot$. They can be allocated

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

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

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

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

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

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

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

Figure 16: The Actris ghost theory.

together (HO-GHOST-ALLOC), are always required to hold the same protocol (HO-GHOST-AGREE), and can only be updated together (HO-GHOST-UPDATE). 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 § 6.8, the rule HO-GHOST-AGREE only gives the equality between the protocols under a later modality ($\triangleright$). The Actris ghost theory connectives are then defined as:

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

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

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

The authority $\mathsf{prot\_ctx}\ (\gamma_1,\gamma_2)\ \vec{v}_1\ \vec{v}_2$ asserts that the buffers $\vec{v}_1$ and $\vec{v}_2$ are consistent with respect to the protocols prot_1 and prot_2 (via $\mathsf{prot\_consistent}\ \vec{v}_1\ \vec{v}_2\ \mathit{prot}_1\ \mathit{prot}_2$). It also asserts the higher-order authoritative ownership $\gamma_1 \mapsto_\bullet \mathit{prot}_1$ and $\gamma_2 \mapsto_\bullet \mathit{prot}_2$ of both protocols. The tokens assert higher-order fragmented ownership $\gamma_1 \mapsto_\circ \mathit{prot}_l'$ and $\gamma_2 \mapsto_\circ \mathit{prot}_r'$ of protocols prot_l' and prot_r' that are weaker than the protocol arguments prot_l and prot_r (via $\mathit{prot}_l' \sqsubseteq \mathit{prot}_l$ and $\mathit{prot}_r' \sqsubseteq \mathit{prot}_r$). Explicit weakening under the subprotocol relation may seem redundant, as weakening is already accounted for in prot_consistent. However, it allows us to weaken the protocols of the tokens without the presence of the authority as shown by the rules PROTO-$\sqsubseteq$-L and PROTO-$\sqsubseteq$-R in Figure 16. The later modality ($\triangleright$) makes sure that $\mathsf{prot\_own_l}\ (\gamma_1,\gamma_2)\ \mathit{prot}$ and $\mathsf{prot\_own_r}\ (\gamma_1,\gamma_2)\ \mathit{prot}$ are contractive in prot. For readability we condense the ghost state identifiers (γ_1,γ_2) into a single identifier γ from now on.

With these definitions at hand, we prove the rules of the ghost theory presented in Figure 16. 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. These are proved through a combination

```
new_chan () ≜ let (l, r, lk) = (new_list (), new_list (), new_lock ()) in
              ((l, r, lk), (r, l, lk))
   send c v ≜ let (l, r, lk) = c in
              acquire lk;
                snoc l v; skipN |r|;
              release lk
try_recv c ≜ let (l, r, lk) = c in
              acquire lk;
                let ret = (if (is_nil r) then (inj₁ ()) else (inj₂ (pop l))) in
              release lk; ret
    recv c ≜ match (try_recv c) with
               inj₁ () ⇒ recv c
             | inj₂ v  ⇒ v
             end
```

Figure 17: Implementation of bidirectional channels in HeapLang.

of the rules for higher-order ghost state from Figure 15, and the rules for the protocol consistency relation prot_consistent from § 6.3.

6.5. **Semantics of channels.** Since Iris is parametric in the programming language that is used, there are various 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 use HeapLang, the default language shipped with Iris, and implement bidirectional channels using a pair of mutable linked lists protected by a lock. Although this implementation is not efficient, contrary to *e.g.,* the implementation by Bizjak et al. [2019], it has the benefit that it gives a clear declarative semantics that corresponds exactly to the intuitive semantics of channels described in § 2.1.

Our implementation of bidirectional channels in HeapLang is displayed in Figure 17. New channels are created by the new_chan function, which allocates two empty mutable linked lists l and r using new_list (), 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.

Values are sent over a channel endpoint (l, r, lk) using the send function. This 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 snoc l v. The skipN $|r|$ instruction is a no-op that is inserted to aid the proof. We come back to the reason why this instruction is needed in § 6.6.

Values are received over a channel endpoint (l, r, lk) using the `send` function, which performs 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 check whether the other endpoint's buffer is empty using `is_nil` r. If it is empty, nothing is returned (*i.e.,* $\texttt{inj}_1\,()$), while otherwise the value is dequeued and returned (*i.e.,* $\texttt{inj}_2\,(\texttt{pop}\ l)$).

6.6. **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:

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

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 $\mathsf{prot\_own_l}\ \gamma\ \mathit{prot}$ or $\mathsf{prot\_own_r}\ \gamma\ \mathit{prot}$ for the corresponding side. Iris's lock representation predicate $\mathsf{is\_lock}$ (previously presented in § 4) 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 $\mathsf{prot\_ctx}\ \gamma\ \vec{v}_1\ \vec{v}_2$, 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 1. 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 16) 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 16 contain a number of later modalities ($\triangleright$) proportional to the other endpoint's buffer in their premise. As explained in § 6.3 these later modalities are the consequence of having to perform a number of inversions 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` $|r|$ 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 appears more often in work on step-indexing, see *e.g.,* [Svendsen et al. 2016; Giarrusso et al. 2020]. It is needed due to the fact that 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. Svendsen et al. [2016] proposed a more liberal version of step-indexed, called *transfinite step-indexing*, to avoid this problem. However, transfinite step-indexing is not available in Iris.

6.7. **Adequacy of Actris.** Having constructed the model of Actris in Iris, we now obtain the following main result:

Theorem 1 Adequacy of Actris**.** *Let ϕ be a first-order predicate over values and suppose the Hoare triple $\{\mathsf{True}\}\, e\, \{\phi\}$ is derivable in Actris, then:*

- ***(Safety):** The program e will not get stuck.*
- ***(Postcondition validity):** If the main thread of e terminates with a value v, then the postcondition $\phi\ v$ holds at the meta-level.*

Since Actris is an internal logic embedded in Iris, the proof is an immediate consequence of Iris's adequacy theorem [Krebbers et al. 2017a; Jung et al. 2018b]. Finally, note that safety implies *session fidelity*—any message that is received has in fact been sent.

6.8. **Solving the recursive domain equation for protocols.** Recall the recursive domain equation for dependent separation protocols from § 6.1:

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

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 15, 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 $\mathsf{iProto}(X^-, X^+)$.

7. 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 (§ 7.1), and go through the full proof of a message-passing program (§ 7.2) and a subprotocol relation (§ 7.3) showcasing the tactics for Actris. During the section we display proofs and proof states taken directly from the Coq mechanisation, which differs in notation from the paper as follows:

	Paper	**Coq Mechanisation**
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`

7.1. **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, the default concurrent language shipped with Iris, 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 that provides support for the iProto type, the subprotocol relation $prot_1 \sqsubseteq prot_2$, the $c \rightarrowtail prot$ connective, the various operations on protocols, and the proof rules as lemmas, 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 18.

7.2. **Tactic support for session type-based reasoning.** To carry out interactive Actris's proof using symbolic execution, we follow the methodology described in the original Iris Proof Mode paper [Krebbers et al. 2017b], which in particular 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.
- Find a corresponding $c \rightarrowtail prot$ hypothesis in the separation logic context.
- Normalise the protocol $prot$ using the rules for duals, composition, recursion, and swapping so it has a $!\, \vec{x}{:}\vec{\tau}\, \langle v \rangle \{P\}.\, prot$ or $?\vec{x}{:}\vec{\tau}\, \langle v \rangle \{P\}.\, prot$ 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 $

Component	**Sections**	**~LOC**
The Actris model	§ 6.1–§ 6.4	1500
Channel implementation and proof rules	§ 6.5–§ 6.6	350
Tactics for symbolic execution	§ 7.2	500
Utilities (linked lists, permutations, *etc.*)	n.a.	450
Authoritative contribution ghost theory	§ 4	150
Recursive domain equation theory solver	§ 6.8	100
Examples:		
• Basic examples	§ 1 and § 4.1	400
• Coarse-grained distributed merge sort	§ 2.3–§ 2.7	250
• Fine-grained distributed merge sort	§ 2.8	300
• Mapper with swapping	§ 3.2	400
• List reversal	§ 3.3	100
• Distributed mapper	§ 4.2	200
• Distributed map-reduce	§ 5	300
Total		5000

Figure 18: Overview of the Actris Coq mechanisation.

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:

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

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:

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

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

```
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 19: Proof of message-passing program

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 19. The proved 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. We start the proof on line 3 by introducing the postcondition continuation Φ, and the hypothesis `HΦ: Φ #42`, and then evaluate the lambda expression with `wp_lam`. On line 4 we apply the specification `start_chan_spec` 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.

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 following intermediate 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. For brevity's sake we refer to the recursive program as `prog_rec`, in the following proof states.

```
"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 }}
```

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 `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 Actris's 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.

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 then resolve the first send operation with Actris's 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 Actris's 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 to resolve the remaining computations. On line 12 we finally close the proof with the continuation hypothesis `by iApply "HΦ"`.

7.3. **Tactic support for subprotocols.** While the Actris tactics automatically apply the subprotocol rules during symbolic execution, as shown in § 7.2, we sometimes want to prove subprotocol relations as explicit lemmas. We have tactic support for such proofs as well, which is integrated with the existing MoSeL tactics `iIntros`, `iExists`, `iFrame`, `iModIntro`, and `iSplitL/iSplitR` by automatically using the subprotocol rules to turn the goal into a 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 ⊑-SEND-IN and ⊑-RECV-IN, 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 ⊑-SEND-OUT and ⊑-RECV-OUT 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 ⊑-SEND-OUT and ⊑-RECV-OUT, and then tries to solve the payload predicate subgoal using `"H"`.

```
1  Lemma list_rev_subprot :
2    ⊢ (<! (l : loc) (vs : list val)> MSG #l {{ llist l vs }};
3       <?> MSG #() {{ llist internal_eq l (reverse vs) }}; END) ⊑
4      (<! (l : loc) (xs : list T)> MSG #l {{ llistI IT l xs }};
5       <?> MSG #() {{ llistI IT l (reverse xs) }}; END).
6  Proof.
7    iIntros (l xs) "Hl".
8    iDestruct (Hlr with "Hl") as (vs) "[Hl HIT]".
9    iExists l, vs. iFrame "Hl".
10   iModIntro. iIntros "Hl".
11   iSplitL.
12   { rewrite big_sepL2_reverse_2. iApply Hlr.
13     iExists (reverse vs). iFrame "Hl HIT". }
14   done.
15 Qed.
```

Figure 20: Proof of subprotocol relation

- `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 rules $\sqsubseteq$-SEND-OUT and $\sqsubseteq$-RECV-OUT, 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 in [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 § 3.3:

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

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:

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

The full Coq proof is shown in Figure 20. On line 7 we start the proof 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-IN, 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 rule ⊑-SEND-OUT, which makes the goal start with `∃(l : loc) (vs : list val)`, so the existentials can be instantiated as usual. To resolve the payload predicate obligation `llist l vs`, we use `iFrame "Hl"`. This uses the ⊑-SEND-OUT 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 ⊑-SEND-MONO to step over the sends, and then introduces the later modality (▷). 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 ⊑-RECV-IN:

```
"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-OUT, 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.

8. Related work

As Actris combines results from both the separation logic and session types community, there is an abundance of related work. This section briefly elaborates on the relation to message passing in separation logic (§ 8.1) and process calculi (§ 8.2), session types (§ 8.3), endpoint sharing (§ 8.4), and verification efforts of map-reduce (§ 8.5).

8.1. **Message passing and separation logic.** Lozes and Villard [2012] proposed a logic, based on previous work by Villard et al. [2009], to reason about programs written in a small imperative language with message passing using channels similar to ours. Messages are labelled, and protocols are handled with a combination of finite-state automata (FSA) with correspondingly labelled transitions and predicates associated with each state of the automata. This combination is similar to, but less general than, STSs in Iris. Their language does not support higher-order functions or delegation, but since their language is restricted to structured concurrency (*i.e.,* not fork-based) and their logic is linear (*i.e.,* not affine), they ensure that all resources like channels and memory are properly deallocated.

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 via message-passing, delegation through higher-order channels, 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 (or binders by construction) as a part of their 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 [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 only 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 did 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] take yet another direction and 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 the channels are implemented using references on the heap. This work is also done in Iris. 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.

The SteelCore framework by [Swamy et al. 2020] is an extensible concurrent separation logic based in F*, which has been used to encode unidirectional synchronous channels. The channels are encoded as a shallow embedding, which is tied together with ghost state to follow a protocol, using a trace of the communicated messages. 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. Their channels are first-class and can also be transferred, effectively achieving delegation. They have postulated that bidirectional asynchronous communication is possible, but have not yet done that. Finally, their protocols do not include higher-order protocol-level logical variables, or any notion of subtyping.

8.2. **Separation logic and process calculi.** Another approach is to verify message-passing programs written in some dialect of process calculus. We focus on related work that combines process calculus with separation logic. 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.

8.3. **Session types.** Seminal work on linear type systems for the pi calculus by Kobayashi et al. [1996] led to the creation of binary session types by Honda et al. [1998].

Bocchi et al. [2010] pushed the boundaries of what can be verified with multi-party 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.

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 the Actor model in existing languages, but do not target functional correctness.

Thiemann and Vasconcelos [2020] introduced label dependent session types, where tails of protocols can depend on the communicated message, which allows for encoding of the choice operators using send and receive. This is similar to our encoding of the choice operators in terms of Actris's dependent send and receive (§ 2.5).

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 recursive operator. Gay et al. [2020] provide further insights on this problem, although they investigate duality rather than subtyping. To reason about recursive types, 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.

8.4. **Endpoint sharing.** One of the key features of session types is that endpoints are owned by a single process. While these endpoints can be delegated (*i.e.,* transferred from one process to another), they typically cannot be shared (*i.e.,* be accessed by multiple

processes concurrently). However, as we demonstrate in § 4, sharing channels endpoints is often desirable, and possible in Actris.

In the pi 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.

8.5. **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.

9. Conclusion and future work

In this paper, we have given a comprehensive account of the Actris logic, which incorporates a protocol mechanism based on session types into concurrent separation logic to enable functional correctness proofs of programs that combine message-passing with other programming and concurrency paradigms. Considering the rich literature on session types and concurrent separation logic, we expect there to be many promising directions for future work.

One of the most prominent extensions of binary session types is multi-party session types [Honda et al. 2008], often called choreographies, which allow concise specifications of message transfers between more than two parties. It would be interesting to explore a multi-party version of dependent separation protocols, similar to the multi-party version of session logic by Costea et al. [2018], to allow Actris to more readily verify programs that make use of multi-party communication.

In addition to safety (*i.e.,* session fidelity), conventional session type systems guarantee properties like deadlock and resource-leak freedom. Since Actris is an extension of concurrent separation logic that supports reasoning about several concurrency primitives and not only message passing, ensuring deadlock freedom is hard. The only prior work in this direction that we are aware of is by Hamin and Jacobs [2019] and Craciun et al. [2015], but it is not immediately obvious how to integrate that with Iris or Actris. Resource-leak freedom has been studied in Iron, an extension of Iris by Bizjak et al. [2019], which makes it possible to prove resource-leak freedom of non-structured fork-based concurrent programs. It would be interesting to build dependent separation protocols on top of Iron instead of Iris.

Acknowledgments

We thank the anonymous reviewers of the POPL'20 paper 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, related either to the POPL'20 paper or this paper. The third author (Robbert Krebbers) was supported by the Dutch Research Council (NWO), project 016.Veni.192.259.

References

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.

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.

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.

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.

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

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.

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. 2020a. Actris: Session-type Based Reasoning in Separation Logic. *PACMPL* 4, POPL (2020), 6:1–6:30.

Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. 2020b. Coq Mechanization of Actris. Available online at `https://gitlab.mpi-sws.org/iris/actris/-/tree/lmcs`.

Jonas Kastberg Hinrichsen, Daniël Louwrink, Jesper Bengtson, and Robbert Krebbers. 2020c. Machine Checked Semantic Session Typing. (2020). Manuscript under submission.

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.

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, 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.

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.

Dimitris Mostrous and Vasco Thudichum Vasconcelos. 2014. Affine Sessions. In *COORDINATION*. 115–130.

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.

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.

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

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