Actris: Session-Type Based Reasoning in Separation Logic

JONAS KASTBERG HINRICHSEN, IT University of Copenhagen, Denmark
JESPER BENGTSON, IT University of Copenhagen, Denmark
ROBBERT KREBBERS, Delft University of Technology, The Netherlands

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 relatively simple specifications. Soundness of Actris is proved using a model of its protocol mechanism in the Iris framework. We mechanised the theory of Actris, together with tactics for symbolic execution of programs, as well as all examples in the paper, in the Coq proof assistant.

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

Additional Key Words and Phrases: Message passing, actor model, concurrency, session types, Iris

ACM Reference Format:
Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. 2020. Actris: Session-Type Based Reasoning in Separation Logic. *Proc. ACM Program. Lang.* 4, POPL, Article 6 (January 2020), 29 pages. https://doi.org/10.1145/3371074

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 many 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.

While message passing is a useful abstraction, it is by no means 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.

Authors' addresses: Jonas Kastberg Hinrichsen, IT University of Copenhagen, Denmark, jkas@itu.dk; Jesper Bengtson, IT University of Copenhagen, Denmark, jebe@itu.dk; Robbert Krebbers, Delft University of Technology, The Netherlands, mail@robbertkrebbers.nl.

2475-1421/2020/1-ART6
https://doi.org/10.1145/3371074

For 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. 2016, 2018b, 2015; Krebbers et al. 2017a]. 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 delegating linear resources, sharing channels over multiple concurrent threads using locks, and more.

1.1 Message Passing in Concurrent Separation Logic

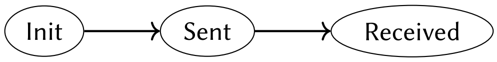
Proving functional correctness of concurrent programs is notoriously difficult, but significant progress has been made since the seminal development of concurrent separation logic by O'Hearn [2004] and Brookes [2004], which made possible expressive frameworks like TaDA [da Rocha Pinto et al. 2014], iCAP [Svendsen and Birkedal 2014], Iris [Jung et al. 2015], FCSL [Nanevski et al. 2014], and VST [Appel 2014] for reasoning about concurrency.

The primary focus of concurrent separation logic and its descendants has been on reasoning about shared-memory concurrency, whereas only relatively little work targets message-passing concurrency. Recently, however, there have been two developments that use separation logic to reason about message passing over networks, where all data must be serialized and messages can be lost and delivered out of order: the Disel logic by Sergey et al. [2018] and the Aneris logic by Krogh-Jespersen et al. [2019]. Programs that operate on the network level have a low level of abstraction and are very different from those written in high-level languages like Erlang, Elixir, Go, and Scala, where messages can contain many types of data, including functions, references, and even channel endpoints. There also exists work that does not use separation logic but that does target message passing in high-level languages *e.g.,* [Francalanza et al. 2011; Lozes and Villard 2012; Oortwijn et al. 2016]. However, none of the work mentioned above provide integration with other programming and concurrency paradigms.

An exception is Tassarotti et al. [2017], who used separation logic to verify a small compiler of a message-passing language into a functional language where channel buffers are modelled on the heap. However, their work and other existing approaches [Jung et al. 2015; Krogh-Jespersen et al. 2019; Sergey et al. 2018] make use of some form of State Transition Systems (STS) to model interaction between processes. 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 being 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 Dependent Separation Protocols

In this paper we take a different route. Instead of using STSs, we extend 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 example above would have the types $c : \,?\mathbb{Z}.\mathbf{end}$ and $c' : \,!\mathbb{Z}.\mathbf{end}$, where $!T$ and $?T$ denotes that a value of type T is sent or received, respectively. Moreover, the types of c and c' are *duals*—when one does a send the other does a receive, and vice versa. From these types one can describe the sequence and type of the data that is passed around over the channels.

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*. There has been some work by Bocchi et al. [2010] to attach logical predicates to session types to say more about the payloads, however their logic is first-order and a lot of effort is put into keeping type checking decidable. Actris, the logic introduced in this paper, makes no such constraints and ports session types into the separation logic directly in the form of a construct $c \rightarrowtail \mathit{prot}$, which denotes ownership of a channel c with dependent separation protocol prot. A dependent separation protocol is a stream consisting of $\mathbf{!}\,\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot}$ and $\mathbf{?}\,\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot}$ nodes, where 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 $\vec{x}:\vec{\tau}$ binds the variables in v, P, and prot. The dependent separation protocols for the above example are:

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

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: $\{\mathsf{True}\}\ \mathit{prog}1\ \{v.\, v = 42\}$, where v is the result of the evaluation.

While this example could also have been type-checked using the formalism of Bocchi et al. [2010], the following stateful example cannot. Here the program stores the value 42 in the heap and sends a reference to it over the channel:

$$\mathit{prog}_2 \triangleq \mathtt{let}\ (c, c') = \mathtt{new\_chan}\ ()\ \mathtt{in}\ \mathtt{fork}\ \{\mathtt{send}\ c'\ (\mathtt{ref}\ 42)\}\,;\, !\,(\mathtt{recv}\ c)$$

This program has the same specification as $prog_1$ but the dependent separation protocols differ:

$$c \rightarrowtail \mathbf{?}\,\ell\,\langle \ell \rangle\{\ell \mapsto 42\}.\,\mathbf{end} \quad \text{and} \quad c' \rightarrowtail \mathbf{!}\,\ell\,\langle \ell \rangle\{\ell \mapsto 42\}.\,\mathbf{end}$$

This protocol denotes that the endpoints exchange a reference ℓ, as well as a *points-to* connective $\ell \mapsto 42$ that describes the ownership and value of the reference ℓ. To perform the exchange, c' then has to *give up* ownership of the location, while c *acquires* it—which is why it can then safely dereference the received location to obtain the expected value.

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

$$\begin{aligned} prog_3 \triangleq\ & \texttt{let}\ (c_1, c_1') = \texttt{new\_chan}\ ()\ \texttt{in} \\ & \texttt{fork}\ \{\texttt{let}\ (c_2, c_2') = \texttt{new\_chan}\ ()\ \texttt{in send}\ c_1'\ c_2; \texttt{send}\ c_2'\ (\texttt{ref}\ 42)\}\,; \\ & !\,(\texttt{recv}\ (\texttt{recv}\ c_1)) \end{aligned}$$

The specification of the program remains the same as $prog_1$ and $prog_2$, while the dependent separation protocols change as follows (we omit the dual protocols for the endpoints c_1' and c_2'):

$$c_1 \rightarrowtail \mathbf{?}\,c\,\langle c \rangle\{c \rightarrowtail \mathbf{?}\,\ell\,\langle \ell \rangle\{\ell \mapsto 42\}.\,\mathbf{end}\}.\,\mathbf{end} \quad \text{and} \quad c_2 \rightarrowtail \mathbf{?}\,\ell\,\langle \ell \rangle\{\ell \mapsto 42\}.\,\mathbf{end}$$

The protocol states that the exchanged value must be a channel endpoint with the specified protocol, meaning that c_1' must give up the ownership of the channel endpoint c_2 thereby delegating it.

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 quantified variables $\vec{x} : \vec{\tau}$. A sample program showing the use of such dependency is:

$$\begin{aligned} prog_4 \triangleq\ & \texttt{let}\ (c, c') = \texttt{new\_chan}\ ()\ \texttt{in} \\ & \texttt{fork}\ \{\texttt{let}\ x = \texttt{recv}\ c'\ \texttt{in send}\ c'\ (x + 2)\}\,; \\ & \texttt{send}\ c\ 40; \texttt{recv}\ c \end{aligned}$$

This program exchanges a value, adds 2 to it and exchanges it again. As previously, the program specification remains the same, while the dependent separation protocol is defined as:

$$c \rightarrowtail \mathbf{!}\,x\,\langle x \rangle\{\mathrm{True}\}.\,\mathbf{?}\,\langle x + 2 \rangle\{\mathrm{True}\}.\,\mathbf{end}$$

This protocol explicitly states that the second exchanged value is exactly the first with 2 added to it.

These dependencies are not limited to first-order data, but can also be used in combination with functions. Consider:

$$\begin{aligned} prog_5 \triangleq\ & \texttt{let}\ (c, c') = \texttt{new\_chan}\ ()\ \texttt{in} \\ & \texttt{fork}\ \{\texttt{let}\ f = \texttt{recv}\ c'\ \texttt{in send}\ c'\ (\lambda\, ().\, f() + 2)\}\,; \\ & \texttt{let}\ r = \texttt{ref}\ 40\ \texttt{in send}\ c\ (\lambda\, ().\ !\,r); \texttt{recv}\ c\ () \end{aligned}$$

This program, like the one before it, exchanges a value to which 2 is added, but postpones the evaluation by wrapping the computation in a closure. Like before, the program specification remains the same, but the dependent separation protocol changes to:

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

The $\mathbf{!}$ 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. 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.

While it has not been captured in the above examples, protocols are closed under composition, branching, and *guarded* recursion. It is worth noting that using dependent recursive protocols, one can keep track of a history of what actions have been performed, which, as will be shown, is especially useful when combining channels with locks.

Although Actris is loosely based on session types, its focus is on proving functional correctness of programs that combine message passing with other paradigms. To enable such proofs, the protocols in Actris are defined using arbitrary separation logic predicates, which means that proof checking is necessarily undecidable—Actris is thus *not* a type system. Furthermore, as a consequence of the focus on functional correctness, Actris does not satisfy properties such as deadlock freedom of session type systems, but it does imply session fidelity through safety (Theorem 5.1).

1.3 Contributions and Outline

This paper introduces **Actris**: a higher-order impredicative concurrent separation logic build on top of the Iris framework for reasoning about functional correctness of message-passing programs 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 branching, recursion, and delegation (Section 2).
- 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 (Section 3).
- 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] (Section 4).
- We give a model of dependent separation protocols in the Iris framework. Using this model we obtain safety (*i.e.,* session fidelity) and postcondition validity of our Hoare triples (Section 5).
- We provide a full mechanisation of Actris [Hinrichsen et al. 2019] using the interactive theorem prover Coq. On top of that, we provide tactics for symbolic execution of dependent separation protocols and mechanise all the examples in the paper (Section 6).

2 A TOUR OF ACTRIS

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

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

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 $\texttt{let}\ \_ = e_1\ \texttt{in}\ e_2$.

The language features the usual operations for heap manipulation. New references can be created using `ref` e, dereferenced using $!\,e$, and assigned to using $e_1 \leftarrow e_2$. Concurrency is supported via `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.

The language supports message passing through bidirectional channels, which are represented using pairs of buffers $(\vec{v}_1, \vec{v}_2)$ of unbounded size. The `new_chan` $()$ operation creates a new channel whose buffers are empty, and returns a tuple of endpoints (c_1, c_2). Bidirectionality is obtained by having one endpoint receive from the other's send buffer and vice versa. That means, `send` $c_i\ v$ enqueues the value v in its own buffer, *i.e.,* $\vec{v}_i$, and `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 `send` $c\ v$ will always reduce, while `recv` c will block as long as the receiving buffer is empty.

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 Section 5, Actris is built as a library on top of the Iris framework [Jung et al. 2016, 2018b, 2015; Krebbers et al. 2017a] and thus inherits all features of Iris. For the purpose of this section, no prior knowledge of Iris is expected as the majority 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 (Section 2.4) and to define recursive protocols (Section 2.6). An extensive overview of Iris can be found in [Jung et al. 2018b].

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 so one can define recursive predicates. But moreover, as we will demonstrate in Section 2.6, the constructors $!\,\vec{x}\!:\!\vec{\tau}\ \langle v\rangle\{P\}.\ \mathit{prot}$ and $?\,\vec{x}\!:\!\vec{\tau}\ \langle v\rangle\{P\}.\ \mathit{prot}$ of dependent separation protocols are contractive in the arguments P and *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.

In order 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 may be omitted whenever 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 $\triangleright$ modality

[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.

Grammar:

$$\begin{array}{rll}
\tau, \sigma ::= & x \mid 0 \mid 1 \mid \mathbb{B} \mid \mathbb{N} \mid \mathbb{Z} \mid \mathsf{Type} \mid \forall x : \tau.\, \sigma \mid & \\
& \mathsf{Loc} \mid \mathsf{Chan} \mid \mathsf{Val} \mid \mathsf{Expr} \mid \mathsf{iProp} \mid \mathsf{iProto} \mid \ldots & \\
t, u, P, Q, \mathit{prot} ::= & x \mid \lambda x : \tau.\, t \mid t(u) \mid t(\tau) \mid & \text{(Polymorphic lambda-calculus)} \\
& \mathsf{True} \mid \mathsf{False} \mid P \wedge Q \mid P \vee Q \mid P \Rightarrow Q \mid & \text{(Propositional logic)} \\
& \forall x : \tau.\, P \mid \exists x : \tau.\, P \mid t = u \mid & \text{(Higher-order logic with equality)} \\
& \mu x : \tau.\, t \mid \triangleright P \mid & \text{(Guarded recursion)} \\
& P * Q \mid P \mathrel{-\!\!*} Q \mid \ell \mapsto v \mid \{P\}\, e\, \{v.\, Q\} \mid & \text{(Separation logic)} \\
& c \rightarrowtail \mathit{prot} \mid \overline{\mathit{prot}} \mid \mathit{prot}_1 \cdot \mathit{prot}_2 \mid \mathbf{end} \mid & \\
& !\, \vec{x}{:}\vec{\tau}\, \langle v \rangle \{P\}.\, \mathit{prot} \mid ?\, \vec{x}{:}\vec{\tau}\, \langle v \rangle \{P\}.\, \mathit{prot} \mid \ldots & \text{(Dependent separation protocols)}
\end{array}$$

Ordinary affine separation logic:

Affine
$$P * Q \Rightarrow P$$

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

Ht-val
$$\{\mathsf{True}\}\, v\, \{w.\, w = v\}$$

Ht-fork
$$\frac{\{P\}\, e\, \{\mathsf{True}\}}{\{P\}\, \mathtt{fork}\ \{e\}\ \{w.\, w = ()\}}$$

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

Recursion:

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

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

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

Heap manipulation:

Ht-alloc
$$\{\mathsf{True}\}\, \mathtt{ref}\, v\, \{\ell.\, \ell \mapsto v\}$$

Ht-load
$$\{\ell \mapsto v\}\ !\, \ell\, \{w.\, w = v \wedge \ell \mapsto v\}$$

Ht-store
$$\{\ell \mapsto v\}\, \ell \leftarrow w\, \{\ell \mapsto w\}$$

Message passing:

$$\{\mathsf{True}\}\, \mathtt{new\_chan}\ ()\, \big\{(c, c').\, c \rightarrowtail \mathit{prot} * c' \rightarrowtail \overline{\mathit{prot}}\big\} \qquad \text{(Ht-newchan)}$$

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

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

Dependent separation protocols:

$$\begin{array}{rcl@{\qquad}rcl}
\overline{!\, \vec{x}{:}\vec{\tau}\, \langle v \rangle \{P\}.\, \mathit{prot}} & = & ?\, \vec{x}{:}\vec{\tau}\, \langle v \rangle \{P\}.\, \overline{\mathit{prot}} & (!\, \vec{x}{:}\vec{\tau}\, \langle v \rangle \{P\}.\, \mathit{prot}_1) \cdot \mathit{prot}_2 & = & !\, \vec{x}{:}\vec{\tau}\, \langle v \rangle \{P\}.\, (\mathit{prot}_1 \cdot \mathit{prot}_2) \\
\overline{?\, \vec{x}{:}\vec{\tau}\, \langle v \rangle \{P\}.\, \mathit{prot}} & = & !\, \vec{x}{:}\vec{\tau}\, \langle v \rangle \{P\}.\, \overline{\mathit{prot}} & (?\, \vec{x}{:}\vec{\tau}\, \langle v \rangle \{P\}.\, \mathit{prot}_1) \cdot \mathit{prot}_2 & = & ?\, \vec{x}{:}\vec{\tau}\, \langle v \rangle \{P\}.\, (\mathit{prot}_1 \cdot \mathit{prot}_2) \\
\overline{\mathbf{end}} & = & \mathbf{end} & \mathbf{end} \cdot \mathit{prot} & = & \mathit{prot} \\
\overline{\overline{\mathit{prot}}} & = & \mathit{prot} & \mathit{prot} \cdot \mathbf{end} & = & \mathit{prot} \\
\overline{\mathit{prot}_1 \cdot \mathit{prot}_2} & = & \overline{\mathit{prot}_1} \cdot \overline{\mathit{prot}_2} & \mathit{prot}_1 \cdot (\mathit{prot}_2 \cdot \mathit{prot}_3) & = & (\mathit{prot}_1 \cdot \mathit{prot}_2) \cdot \mathit{prot}_3
\end{array}$$

Fig. 1. The grammar and a selection of rules of Actris.

in the precondition, which when combined with the Löb rule allows one to reason 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, one can discard arbitrary separation logic resources 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 $c \rightarrowtail \mathit{prot}$ 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 $\mathbf{!}\,\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot}$ and $\mathbf{?}\,\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot}$ 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 quantifiers $\vec{x}:\vec{\tau}$ can be used to bind variables in v, P, and *prot*. For example, $\mathbf{!}\,(b:\mathbb{B})\,(\ell:\mathsf{Loc})\,(i:\mathbb{N})\,\langle(b,\ell)\rangle\{\ell \mapsto i * 10 < i\}.\,\mathit{prot}$ 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 $\overline{\mathit{prot}}$ connective denotes the *dual* of a protocol. As with conventional session types, it transforms the protocol by changing all sends (**!**) into receives (**?**), and vice versa. Taking the dual twice thus results in the original protocol. The connective $\mathit{prot}_1 \cdot \mathit{prot}_2$ *composes* the protocols prot_1 and prot_2, which is achieved by substituting any **end** in prot_1 with prot_2.

The rule Ht-newchan allow ascribing any protocol to newly created channels using `new_chan` (), obtaining ownership of $c \rightarrowtail \mathit{prot}$ and $c' \rightarrowtail \overline{\mathit{prot}}$ for the respective endpoints. The duality of the protocol guarantees that any receive (**?**) is matched with a send (**!**) by the dual endpoint, which is crucial for establishing safety (*i.e.,* session fidelity, see Section 5.4).

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. Concretely, to send a message w, one need to give up ownership of $c \rightarrowtail \mathbf{!}\,\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot}$, pick an appropriate instantiation $\vec{t}$ for the quantified variables $\vec{x}:\vec{\tau}$ so that $w = v[\vec{t}/\vec{x}]$, and give up ownership of the associated resources $P[\vec{t}/\vec{x}]$. Subsequently, one gets back ownership of the protocol tail $\mathit{prot}[\vec{t}/\vec{x}]$.

The rule Ht-recv for `recv` c is essentially dual to the rule Ht-send. One needs to give up ownership of $c \rightarrowtail \mathbf{?}\,\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot}$, and in return gets acquires 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 $\mathit{prot}[\vec{y}/\vec{x}]$, 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.

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

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

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:

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

The channel c follows the following dependent separation protocol:

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

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 $\mathsf{sorted\_of}_R\ \vec{x}\ \vec{y}$ is true iff $\vec{x}$ is a sorted version of $\vec{y}$ with respect to the relation R. The specification of the service and the client is as follows:

$$\begin{array}{l} \{\mathsf{cmp\_spec}\ I\ R\ cmp * c \rightarrowtail \overline{\mathsf{sort\_prot}\ I\ R} \cdot prot\} \\ \quad \mathtt{sort\_service}\ cmp\ c \\ \{c \rightarrowtail prot\} \end{array} \qquad \begin{array}{l} \{\mathsf{cmp\_spec}\ I\ R\ cmp * \ell \mapsto_I \vec{x}\} \\ \quad \mathtt{sort\_client}\ cmp\ \ell \\ \{\exists \vec{y}.\, \mathsf{sorted\_of}_R\ \vec{y}\ \vec{x} * \ell \mapsto_I \vec{y}\} \end{array}$$

There are two important things to note about these specifications. First, the protocol 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* composed 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 Section 2.6 and 2.7.

The proof of these specifications is almost entirely performed by symbolic execution using the rules Ht-newchan, Ht-send, Ht-recv, and the standard separation logic rules.

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

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

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 $\texttt{sort\_service}_{\texttt{func}}$ in Figure 3, which receives the comparison function over the channel instead of via a lambda abstraction. In order to verify this program, we extend the protocol sort_prot by first sending the comparison function and then continuing as previously. Note that the new protocol includes the quantifications of the functional specification of the comparison function cmp, including the polymorphic type T, the relation R, and the interpretation predicate I:

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

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:

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

2.5 Branching

Branching behaviour is common in message-passing communication protocols and is readily available in Actris using dependent separation protocols. Branching is encoded in terms of 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 `left` $\triangleq$ `true` and `right` $\triangleq$ `false` to be used together with `select` for the sake of readability. Due to the higher-order nature of Actris, the usual protocol specifications for branching from session types can be encoded as regular logical branching within the protocols:

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

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

$$\{c \rightarrowtail prot_1\ {}_{\{Q_1\}}{\oplus}_{\{Q_2\}}\ prot_2 * \texttt{if}\ b\ \texttt{then}\ Q_1\ \texttt{else}\ Q_2\}\ \texttt{select}\ c\ b\ \{c \rightarrowtail \texttt{if}\ b\ \texttt{then}\ prot_1\ \texttt{else}\ prot_2\}$$

Ht-branch

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

```
sort_servicerec cmp c ≜                  sort_clientrec cmp l ≜
  branch c with                            let c = start sort_servicerec cmp in
   left   ⇒ sort_service cmp c;            iter (λ l′. select c left; send c l′; recv c) l;
            sort_servicerec cmp c          select c right
  | right ⇒ ()
  end
```

Fig. 4. A recursive version of the sort service that can perform multiple jobs in sequence.

Apart from branching on Boolean values, one can use dependent separation protocols to encode branching on any enumeration type (*e.g.*, lists, natural numbers, days of the week, *etc.*).

2.6 Recursive Protocols

We will now use branching and recursion to verify the correctness of a sorting service that supports performing multiple sorting jobs in sequence. The code of the sorting service $\texttt{sort\_service}_{\texttt{rec}}$ and a possible client $\texttt{sort\_client}_{\texttt{rec}}$ are displayed in Figure 4. The service $\texttt{sort\_service}_{\texttt{rec}}$ contains a loop in which branching is used to either terminate the service, or to sort an individual list using the distributed merge sort algorithm `sort_service` from Section 2.3. The client $\texttt{sort\_client}_{\texttt{rec}}$ 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:

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

The protocol uses the branching 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 dictates that one can proceed 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 $\mu\, x : \tau.\ t$. 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, similar to $\mathbf{?}\,\vec{x}\!:\!\vec{\tau}\,\langle v\rangle\{P\}.\ prot$, is contractive in the arguments P and *prot*.

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

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

$$\{\mathsf{cmp\_spec}\ I\ R\ cmp * \ell \mapsto_J \vec{\vec{x}}\}\ \ \texttt{sort\_client}_{\texttt{rec}}\ cmp\ \ell\ \ \{\exists \vec{\vec{y}}.\ |\vec{\vec{y}}| = |\vec{\vec{x}}| * \ell \mapsto_J \vec{\vec{y}} * (\forall i < |\vec{\vec{x}}|.\ \mathsf{sorted\_of}_R\ \vec{y}_i\ \vec{x}_i)\}$$

We let $J \triangleq \lambda\,\ell'\,\vec{y}.\ \ell' \mapsto_I \vec{y}$ 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 post-composition.

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 $\texttt{sort\_client}_{\texttt{func}}$, or in a way that a different comparison function can be used for each sorting job.

```
sort_service_del cmp c ≜                          sort_client_del cmp l =
  branch c with                                     let c = start sort_service_del cmp in
   left  ⇒                                          let k = new_list () in
      let c′ = start sort_service cmp in            iter (λ l′.select c left;
      send c c′;                                            let c′ = recv c in
      sort_service_del cmp c                                push c′ k; send c′ l′) l
  | right ⇒ ()                                      send c right;
  end                                               iter recv k
```

Fig. 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 protocols in Actris, reasoning about programs that make use of delegation is readily available. The protocols $\mathbf{!}\,\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\,prot$ and $\mathbf{?}\,\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\,prot$ 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 sort_service$_{\texttt{del}}$ variant of the recursive sorting service in Figure 5, which allows one to perform multiple sorting jobs in parallel. To enable parallelism, it delegates a new channel c' to an inner sorting service for each sorting job.

The client sort_client$_{\texttt{del}}$ 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:

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

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

$$\begin{array}{ll}\left\{\begin{array}{l}\mathsf{cmp\_spec}\ I\ R\ cmp\ *\\ c \rightarrowtail \overline{\mathsf{sort\_prot}_{\mathsf{del}}} \cdot prot\end{array}\right\} & \left\{\mathsf{cmp\_spec}\ I\ R\ cmp * \ell \mapsto_J \vec{\vec{x}}\right\}\\ \quad \texttt{sort\_service}_{\texttt{del}}\ cmp\ c & \quad \texttt{sort\_client}_{\texttt{del}}\ cmp\ \ell\\ \{c \rightarrowtail prot\} & \left\{\exists \vec{\vec{y}}.\ |\vec{\vec{y}}| = |\vec{\vec{x}}| * \ell \mapsto_J \vec{\vec{y}} * (\forall i < |\vec{\vec{x}}|.\ \mathsf{sorted\_of}_R\ \vec{\vec{y}}_i\ \vec{\vec{x}}_i)\right\}\end{array}$$

As before, we let $J \triangleq \lambda \ell'\,\vec{y}.\ \ell' \mapsto_I \vec{y}$ to express that ℓ points to a list of lists $\vec{\vec{x}}$. Once again the proofs are straightforward, as it is simply a combination of a 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 sent and received. We demonstrate this feature by considering a fine-grained version of the distributed merge sort service. This version sort_service$_{fg}$, as

```
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 c with
   right ⇒ assert false
  | left  ⇒
    let x = recv c1 in
    merge_fg^aux cmp c x c1 c2
  end

merge_fg^aux 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_fg^aux cmp c y c2 c1
    else
      select c left; send c y;
      merge_fg^aux 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
```

Fig. 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).

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 branching 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 $\texttt{sort\_client}_{\mathit{fg}}$ 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, and puts the received values back into the linked list using the `recv_all` function.

A suitable protocol for proving functional correctness of the fine-grained sorting service is:

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

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

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

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

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

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

$$\begin{array}{ll}\left\{\mathsf{cmp\_spec}\ I\ R\ \mathit{cmp} * c \rightarrowtail \overline{\mathsf{sort\_prot}_{\mathit{fg}}} \cdot \mathit{prot}\right\} & \{\mathsf{cmp\_spec}\ I\ R\ \mathit{cmp} * \ell \mapsto_I \vec{x}\}\\ \quad \mathsf{sort\_prot}_{\mathit{fg}}\ c & \quad \texttt{sort\_client}_{\mathit{fg}}\ \mathit{cmp}\ \ell\\ \{c \rightarrowtail \mathit{prot}\} & \{\exists \vec{y}.\ \ell \mapsto_I \vec{y} * \mathsf{sorted\_of}_R\ \vec{y}\ \vec{x}\}\end{array}$$

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 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

$$\{R\}\ \texttt{new\_lock}\ ()\ \{lk.\ \mathsf{is\_lock}\ lk\ R\} \qquad \text{(Ht-new-lock)}$$
$$\{\mathsf{is\_lock}\ lk\ R\}\ \texttt{acquire}\ lk\ \{R\} \qquad \text{(Ht-acquire)}$$
$$\{\mathsf{is\_lock}\ lk\ R * R\}\ \texttt{release}\ lk\ \{\mathsf{True}\} \qquad \text{(Ht-release)}$$
$$\mathsf{is\_lock}\ lk\ R \twoheadrightarrow \mathsf{is\_lock}\ lk\ R * \mathsf{is\_lock}\ lk\ R \qquad \text{(Lock-dup)}$$

Fig. 7. The rules Actris inherits from Iris for locks.

```
prog_lock ≜ let c = start (λ c. let lk = new_lock () in
                                fork {acquire lk; send c 21; release lk} ;
                                acquire lk; send c 21; release lk) in
            recv c + recv c
```

Fig. 8. A sample program that combines locks and channels to achieve manifest sharing.

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 (Section 3.1). We then consider a more challenging example of a distributed load-balancing mapper (Section 3.2).

3.1 Locks and Ghost State

Using the language from Section 2.1 one can 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 7.

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 7 make use of the representation predicate 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 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 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 8, which uses a lock to share a channel endpoint between two threads that each send the integer 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:

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

$$\begin{aligned}
\mathsf{True} &\Rrightarrow \exists \gamma.\, \mathsf{auth}_\gamma\, 0 && \text{(AUTH-INIT)}\\
\mathsf{auth}_\gamma\, n &\Rrightarrow \mathsf{contrib}_\gamma * \mathsf{auth}_\gamma\, (1+n) && \text{(AUTH-ALLOC)}\\
\mathsf{auth}_\gamma\, (1+n) * \mathsf{contrib}_\gamma &\Rrightarrow \mathsf{auth}_\gamma\, n && \text{(AUTH-DEALLOC)}\\
\mathsf{auth}_\gamma\, n * \mathsf{contrib}_\gamma &\mathrel{-\!\!*} n > 0 && \text{(AUTH-CONTRIB-POS)}
\end{aligned}$$

Fig. 9. The authoritative contribution ghost theory.

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. 2018b, 2015].

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 9. 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 9 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:

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

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 $\overline{\mathtt{lock\_prot}\ n}$ 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 $\overline{\mathtt{lock\_prot}\ n}$ to $\overline{\mathtt{lock\_prot}\ (n-1)}$. Subsequently, the sending threads can deallocate the token $\mathsf{contrib}_\gamma$ (using AUTH-DEALLOC) and restore the lock invariant.

3.2 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

```
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; release lk;   (* send it back *)
            mapper_worker f_v lk c
  end
```

Fig. 10. A worker of the distributed mapper service.

$$\begin{array}{rcll}
\mathsf{True} & \Rrightarrow & \exists \gamma.\, \mathsf{auth}_\gamma\, 0\, \emptyset & (\textsc{AuthM-init}) \\
\mathsf{auth}_\gamma\, n\, X & \Rrightarrow & \mathsf{auth}_\gamma\, (1+n)\, X * \mathsf{contrib}_\gamma\, \emptyset & (\textsc{AuthM-alloc}) \\
\mathsf{auth}_\gamma\, n\, X * \mathsf{contrib}_\gamma\, \emptyset & \Rrightarrow & \mathsf{auth}_\gamma\, (n-1)\, X & (\textsc{AuthM-dealloc}) \\
\mathsf{auth}_\gamma\, n\, X * \mathsf{contrib}_\gamma\, Y & \Rrightarrow & \mathsf{auth}_\gamma\, n\, (X \uplus Z) * \mathsf{contrib}_\gamma\, (Y \uplus Z) & (\textsc{AuthM-add}) \\
Z \subseteq Y * \mathsf{auth}_\gamma\, n\, X * \mathsf{contrib}_\gamma\, Y & \Rrightarrow & \mathsf{auth}_\gamma\, n\, (X \setminus Z) * \mathsf{contrib}_\gamma\, (Y \setminus Z) & (\textsc{AuthM-remove}) \\
\mathsf{auth}_\gamma\, n\, X * \mathsf{contrib}_\gamma\, Y & -\!\!* & n > 0 * Y \subseteq X & (\textsc{AuthM-contrib-agree}) \\
\mathsf{auth}_\gamma\, 1\, X * \mathsf{contrib}_\gamma\, Y & -\!\!* & Y = X & (\textsc{AuthM-contrib-agree1})
\end{array}$$

Fig. 11. The authoritative contribution ghost theory extended with multisets.

over a list. Our distributed mapper consists of one client that distributes the work, and a number of workers that perform the function f on individual elements x 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 10, 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 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 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 is encapsulated by the worker, we omit the code of the client for brevity's sake.[2]

[2]The interested reader can find the entire code in the accompanied Coq development [Hinrichsen et al. 2019].

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

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

The protocol is parameterised by representation predicates I_T and I_U that relate language values to elements of type T and U in the logic, as well as a function $f : T \to \mathsf{List}\ U$ that specifies the behaviour of the language-level function f_v. The connection between f and f_v is formalised as:

$$\begin{array}{l}
\mathsf{f\_spec}\ (I_T : T \to \mathsf{Val} \to \mathsf{iProp})\ (I_U : U \to \mathsf{Val} \to \mathsf{iProp})\ (f : T \to \mathsf{List}\ U)\ (f_v : \mathsf{Val}) \triangleq \\
\quad \forall x\, v.\ \{I_T\ x\ v\}\ f_v\ v\ \{\ell.\ \ell \mapsto_{I_U} f\ x\}
\end{array}$$

Similar to `lock_prot` from Section 3.1, mapper_prot is indexed by the number of remaining workers n. On top of that, it carries a multiset X describing the values currently being processed by all the workers. The multiset X is used to make sure that the returned results are in fact the result of mapping the function f. The condition $(n = 1) \Rightarrow (X = \emptyset)$ on the branch (&) 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 Section 3.1. 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 11. 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 `mapper_worker` and a possible top-level client `mapper_client` that uses n workers to map f_v over the linked list ℓ are as follows:

$$\left\{\begin{array}{l}\mathsf{f\_spec}\ I_T\ I_U\ f\ f_v * \mathsf{contrib}_\gamma\ \emptyset\ * \\ \mathsf{is\_lock}\ lk\ \left(\begin{array}{l}\exists n\, X. \mathsf{auth}_\gamma\, n\, X\ * \\ \qquad c \rightarrowtail \overline{\mathsf{mapper\_prot}\ I_T\ I_U\ f\ n\ X}\end{array}\right)\end{array}\right\} \\ \quad \mathtt{mapper\_worker}\ f_v\ lk\ c \\ \{\mathsf{True}\}$$

$$\left\{\begin{array}{l}\mathsf{f\_spec}\ I_T\ I_U\ f\ f_v\ * \\ 0 < n * \ell \mapsto_{I_T} \vec{x}\end{array}\right\} \\ \quad \mathtt{mapper\_client}\ n\ f_v\ \ell \\ \{\exists \vec{y}.\ \vec{y} \equiv_{\mathrm{p}} \mathtt{flatMap}\ f\ \vec{x} * \ell \mapsto_{I_U} \vec{y}\}$$

The lock invariant and specification of `mapper_worker` are similar to those used in the simple example in Section 3.1. The specification of `mapper_client` $n\ f_v\ \ell$ simplify 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 `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 $\mathsf{mapper\_prot}\ I_T\ I_U\ f\ n$, where n is the initial number of workers. 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.

4 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 (Section 2.8) and the distributed load-balancing mapper (Section 3.2). 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. 2019].

4.1 A Functional Specification of Map-Reduce

The purpose of the map-reduce model is to transform an input set of type List T into an output set of type List V using two functions f (often called "map") and g (often called "reduce"):

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

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):

$$\mathtt{flatMap}\, f \quad : \quad \mathsf{List}\ T \rightarrow \mathsf{List}\ (K * U)$$

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

$$\mathtt{group} \quad : \quad \mathsf{List}\ (K * U) \rightarrow \mathsf{List}\ (K * \mathsf{List}\ U)$$

(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:

$$\mathtt{flatMap}\, g \quad : \quad \mathsf{List}\ (K * \mathsf{List}\ U) \rightarrow \mathsf{List}\ V$$

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

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

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:

$$f : \mathsf{String} \rightarrow \mathsf{List}\ (\mathsf{String} * \mathbb{N}) \triangleq \lambda\, x.\ [(x, 1)]$$
$$g : (\mathsf{String} * \mathsf{List}\ \mathbb{N}) \rightarrow \mathsf{List}\ (\mathsf{String} * \mathbb{N}) \triangleq \lambda\, (k, \vec{n}).\ [(k, \Sigma_{i<|\vec{n}|}.\ \vec{n}_i)]$$

4.2 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 `mapper_worker` from Section 3, parameterised with the f_v function. Additionally start an instance of $\texttt{sort\_service}_{fg}$ from Section 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 been forwarded to the sorting service.
(3) Start m instances of the `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.

4.3 Functional Correctness of Map-Reduce

The specification of the program is as follows:

$$\begin{array}{l}\{0 < n \ast 0 < m \ast \mathsf{f\_spec}\ I_T\ I_{\mathbb{Z}\ast U}\ f\ f_\mathsf{v} \ast \mathsf{f\_spec}\ I_{\mathbb{Z}\ast\mathsf{List}\ U}\ I_V\ g\ g_\mathsf{v} \ast \ell \mapsto_{I_T} \vec{x}\}\\ \quad \texttt{map\_reduce}_\mathsf{v}\ n\ m\ f_\mathsf{v}\ g_\mathsf{v}\ \ell\\ \{\exists \vec{z}.\ \vec{z} \equiv_p \texttt{map\_reduce}\ f\ g\ \vec{x} \ast \ell \mapsto_{I_V} \vec{z}\}\end{array}$$

The $\mathsf{f\_spec}$ predicates (as introduced in Section 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, $I_{\mathbb{Z}\ast U}$, $I_{\mathbb{Z}\ast\mathsf{List}\ U}$, and I_V for the types in question. Lastly, the $\ell \mapsto_{I_T} \vec{x}$ 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 Section 4.1.

5 THE MODEL OF ACTRIS

Actris is defined as an internal logic embedded in the Iris framework [Jung et al. 2016, 2018b, 2015; Krebbers et al. 2017a]. This means that the type iProto of dependent separation protocols and the connective $c \rightarrowtail prot$ for the ownership of a channel endpoint are definitions in the Iris logic, and that the Actris proof rules are lemmas in the Iris logic. In this section we describe the relevant aspects of this embedding. First, we present our definitional semantics of bidirectional channels (Section 5.1). We then show how the type iProto is modelled (Section 5.2) and how the connective $c \rightarrowtail prot$ for channel ownership is defined (Section 5.3). Finally, we show how adequacy (safety and postcondition validity of Hoare triples) follows from the embedding in Iris (Section 5.4).

5.1 Semantics of Channels

Since the Iris framework is parametric in the programming language that is being used, there are various approaches to extend it 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 went for the second approach. We used HeapLang, the default concurrent language shipped with Iris, and implemented bidirectional channels using a pair of linked-lists protected by a lock. Although this implementation is not efficient, it has the benefit that it gives a clear declarative semantics that corresponds exactly to the intuitive semantics given in Section 2.1.

At the level of the logic, the state of both buffers of a bidirectional channel is described using the representation predicate $(c_1, c_2) \rightarrowtail (\vec{v}_1, \vec{v}_2)$. We prove Jacobs and Piessens [2011]-style logically atomic specifications that roughly correspond to the following Hoare triples:

$$\begin{array}{rl}
\{\mathsf{True}\}\ \mathtt{new\_chan}\ ()\ & \{(c_1, c_2).\ (c_1, c_2) \rightarrowtail (\epsilon, \epsilon)\} \\
\{(c_1, c_2) \rightarrowtail (\vec{v}_1, \vec{v}_2)\}\ \mathtt{send}\ c_1\ w\ & \{(c_1, c_2) \rightarrowtail (\vec{v}_1 \cdot [w], \vec{v}_2)\} \\
\{(c_1, c_2) \rightarrowtail (\vec{v}_1, \vec{v}_2)\}\ \mathtt{send}\ c_2\ w\ & \{(c_1, c_2) \rightarrowtail (\vec{v}_1, \vec{v}_2 \cdot [w])\} \\
\{(c_1, c_2) \rightarrowtail (\vec{v}_1, \vec{v}_2)\}\ \mathtt{recv}\ c_1\ & \{w.\ (\vec{v}_2 = [w] \cdot \vec{w}) * (c_1, c_2) \rightarrowtail (\vec{v}_1, \vec{w})\} \\
\{(c_1, c_2) \rightarrowtail (\vec{v}_1, \vec{v}_2)\}\ \mathtt{recv}\ c_2\ & \{w.\ (\vec{v}_1 = [w] \cdot \vec{w}) * (c_1, c_2) \rightarrowtail (\vec{w}, \vec{v}_2)\}
\end{array}$$

In Section 5.4 we show how Actris's rules for channels (Ht-newchan, Ht-send and Ht-recv) are derived from these logically atomic specifications. It is worth pointing out that the embedding of Actris only makes use of these specifications, and thus abstracts from the implementation details of the channel. This means that the channel implementation could be replaced with a more efficient version, or with a version that is natively integrated into the language.

5.2 The Model of Dependent Separation Protocols

Dependent separation protocols are streams of the form $\,!\,\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\,prot$ and $\,?\,\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\,prot$, which describe the expected values v and associated resources P that should be transferred over a bidirectional channel, after which it continues as the tail protocol *prot*. These streams are either infinite or finite, where the finite streams are terminated by an **end** constructor. What makes dependent separation protocols different from ordinary streams (that are typically defined as a coinductive data type), is the sequence of logical quantifiers $\vec{x}:\vec{\tau}$ that bind into the expected value v, the resources P, and the tail protocol *prot*. The quantifiers $\vec{x}:\vec{\tau}$ are higher-order and impredicative, meaning that the types $\vec{\tau}$ can be any type of Iris, including functions, propositions, predicates, and protocols themselves.

In order to give a model of dependent separation protocols, we first need to consider what $\mathbf{!}\,\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot}$ and $\mathbf{?}\,\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot}$ mean semantically. As the quantifiers $\vec{x}:\vec{\tau}$ bind into v P, and *prot*, these protocols can be thought of as predicates over the transferred physical value and actual tail of the protocol. Since dependent separation protocols describe the logical resources that are being transferred along with the message, they should be modelled using Iris predicates (iProp) instead of meta-level predicates (Prop). This leads to the following formal intermediate definition:

$$\begin{aligned}\widehat{\mathsf{iProto}} &= 1 + (\mathsf{action} \times (\mathsf{Val} \to \blacktriangleright\widehat{\mathsf{iProto}} \to \mathsf{iProp})) \qquad (\text{where action} ::= \mathbf{send} \mid \mathbf{recv})\\ \mathbf{end} &\triangleq \mathtt{inj}_1\,()\\ \mathbf{!}\,\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot} &\triangleq \mathtt{inj}_2\,(\mathbf{send}, \lambda\, w\, \mathit{prot}'.\, \exists \vec{x}:\vec{\tau}.\, (v = w) * (\triangleright P) * (\mathit{prot}' = \mathsf{next}\ \mathit{prot}))\\ \mathbf{?}\,\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot} &\triangleq \mathtt{inj}_2\,(\mathbf{recv}, \lambda\, w\, \mathit{prot}'.\, \exists \vec{x}:\vec{\tau}.\, (v = w) * (\triangleright P) * (\mathit{prot}' = \mathsf{next}\ \mathit{prot}))\end{aligned}$$

We define $\widehat{\mathsf{iProto}}$ using Iris's support for solving *guarded recursive domain equations* [America and Rutten 1989; Birkedal et al. 2012]. This means that the recursive occurrence of $\widehat{\mathsf{iProto}}$ must appear below a *later* ($\blacktriangleright$) operator (whose only constructor is $\mathsf{next} : T \to \blacktriangleright T$). The left part of the sum type indicates that the protocol has terminated, while the right part is a predicate that describes the exchange. The existential quantifiers in the definitions make sure that the variables $\vec{x}:\vec{\tau}$ bind into v, P, and *prot*. Moreover, to make it possible to construct recursive protocols, $\mathbf{!}\,\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot}$ and $\mathbf{?}\,\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot}$ are contractive in P and *prot* due to the uses of $\triangleright P$ and $\mathsf{next}\ \mathit{prot}$.

Since Iris's ghost theory will be instantiated with iProto in Section 5.3, iProto and iProp need to be defined in a mutually recursive fashion. Unfortunately, as a consequence of this, along with the negative occurrence of iProto, the above definition of $\widehat{\mathsf{iProto}}$ cannot be constructed in the current Coq development of Iris without significant manual effort. Although there is no fundamental restriction preventing this definition [Birkedal et al. 2012, § 7], the necessary changes to the Coq development needed to construct such types are orthogonal to the purpose of this paper. As such, the actual—albeit less intuitive—definition of iProto is as follows:

$$\begin{aligned}\mathsf{iProto} &= 1 + (\mathsf{action} \times (\mathsf{Val} \to (\blacktriangleright\mathsf{iProto} \to \mathsf{iProp}) \to \mathsf{iProp}))\\ \mathbf{end} &\triangleq \mathtt{inj}_1\,()\\ \mathbf{!}\,\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot} &\triangleq \mathtt{inj}_2\,(\mathbf{send}, \lambda\, w\, (f : \blacktriangleright\mathsf{iProto} \to \mathsf{iProp}).\, \exists \vec{x}:\vec{\tau}.\, (v = w) * \triangleright P * f(\mathsf{next}\ \mathit{prot}))\\ \mathbf{?}\,\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot} &\triangleq \mathtt{inj}_2\,(\mathbf{recv}, \lambda\, w\, (f : \blacktriangleright\mathsf{iProto} \to \mathsf{iProp}).\, \exists \vec{x}:\vec{\tau}.\, (v = w) * \triangleright P * f(\mathsf{next}\ \mathit{prot}))\end{aligned}$$

In this definition, we use a predicate $f : \blacktriangleright\mathsf{iProto} \to \mathsf{iProp}$ to restrict the tail protocol, causing iProto to appear in a positive position. When using these protocols in Section 5.3, we always pick the predicate f to be $(\lambda\, \mathit{prot}''.\, \mathit{prot}' = \mathit{prot}'')$ where prot' is the actual tail, which would otherwise have been supplied as the tail argument to the ideal definition. Hence, the definition iProto is functionally equivalent to the ideal definition $\widehat{\mathsf{iProto}}$.

With the actual definition at hand, the dual $\overline{(\_)}$ and composition $(\_ \cdot \_)$ operations are defined using Iris's guarded recursion operator $(\mu\, x : \tau.\, t)$ as follows:

$$\overline{(\_)} \triangleq \mu\, \mathit{rec}.\, \lambda\, \mathit{prot}.\, \begin{cases}\mathtt{inj}_1\,() & \text{if } \mathit{prot} = \mathtt{inj}_1\,()\\ \mathtt{inj}_2\,(\overline{a}, \lambda\, v\, f.\, \Phi\ v\ (f \circ \mathsf{map}\ \mathit{rec})) & \text{if } \mathit{prot} = \mathtt{inj}_2\,(a, \Phi)\end{cases}$$

$$(\_ \cdot \mathit{prot}_2) \triangleq \mu\, \mathit{rec}.\, \lambda\, \mathit{prot}_1.\, \begin{cases}\mathit{prot}_2 & \text{if } \mathit{prot}_1 = \mathtt{inj}_1\,()\\ \mathtt{inj}_2\,(a, \lambda\, v\, f.\, \Phi\ v\ (f \circ \mathsf{map}\ \mathit{rec})) & \text{if } \mathit{prot}_1 = \mathtt{inj}_2\,(a, \Phi)\end{cases}$$

Here, we let $\overline{\mathbf{send}} \triangleq \mathbf{recv}$, $\overline{\mathbf{recv}} \triangleq \mathbf{send}$, and $\mathsf{map} : (T \to U) \to (\blacktriangleright T \to \blacktriangleright U)$, which is the mapping function on the $\blacktriangleright$-type.

$$\mathsf{True} \Rrightarrow \exists \gamma.\, (\gamma \mapsto_{\bullet} \mathit{prot}) * (\gamma \mapsto_{\circ} \mathit{prot}) \qquad (\textsc{HO-ghost-alloc})$$

$$(\gamma \mapsto_{\bullet} \mathit{prot}) * (\gamma \mapsto_{\circ} \mathit{prot}') \Rightarrow \triangleright(\mathit{prot} = \mathit{prot}') \qquad (\textsc{HO-ghost-agree})$$

$$(\gamma \mapsto_{\bullet} \mathit{prot}) * (\gamma \mapsto_{\circ} \mathit{prot}') \Rrightarrow (\gamma \mapsto_{\bullet} \mathit{prot}'') * (\gamma \mapsto_{\circ} \mathit{prot}'') \qquad (\textsc{HO-ghost-update})$$

Fig. 12. Higher-order ghost variables in Iris.

5.3 The Model of Channel Ownership

Now that we have given a semantics of channels (Section 5.1) and a model of dependent separation protocols (Section 5.2), it remains to use these concepts to define the $c \rightarrowtail \mathit{prot}$ connective.

In order to define Actris's connectives $c_1 \rightarrowtail \mathit{prot}_1$ and $c_2 \rightarrowtail \mathit{prot}_2$ for ownership of the endpoints c_1 and c_2 of a bidirectional channel, we need to tie prot_1 and prot_2 to the physical contents of the channel buffers as captured by the representation predicate $(c_1, c_2) \rightarrowtail (\vec{v}_1, \vec{v}_2)$. To obtain duality of the endpoint protocols, they must always be in one of two configurations:

- There are zero or more messages $\vec{v}_1$ in transit from the first endpoint to the second. That is, the buffer contents is $(c_1, c_2) \rightarrowtail (\vec{v}_1, \epsilon)$, and prot_2 starts with a series of receive (?) nodes that match up with the contents $\vec{v}_1$ of the first buffer.
- There are zero or more messages $\vec{v}_2$ in transit from the second endpoint to the first. That is, the buffer contents is $(c_1, c_2) \rightarrowtail (\epsilon, \vec{v}_2)$, and prot_1 starts with a series of receive (?) nodes that match up with the contents $\vec{v}_2$ of the second buffer.

Note that dependent separation protocols (and binary session types) enforce that either of the two channel buffers is always empty as a channel endpoint can only start sending messages (!) once it has received (?) all messages sent by the other endpoint. This property is not generally true for bidirectional channels.

This informal invariant can be formalised in Iris using the definition I, which, in turn, is used to define the connective $c \rightarrowtail \mathit{prot}$ for channel ownership:

$$\begin{aligned}
\mathsf{interp}\ \epsilon\ \mathit{prot}_1\ \mathit{prot}_2 &\triangleq (\mathit{prot}_1 = \overline{\mathit{prot}_2})\\
\mathsf{interp}\ ([v] \cdot \vec{v})\ \mathit{prot}_1\ \mathit{prot}_2 &\triangleq \exists \Phi\ \mathit{prot}_2'.\ (\mathit{prot}_2 = \mathtt{inj}_2\ (\mathbf{recv}, \Phi)) *\\
&\qquad \Phi\ v\ (\lambda\, \mathit{prot}''.\ \mathsf{next}\ \mathit{prot}_2' = \mathit{prot}'') *\\
&\qquad \triangleright \mathsf{interp}\ \vec{v}\ \mathit{prot}_1\ \mathit{prot}_2'\\
I\ \gamma_1\ \gamma_2\ c_1\ c_2 &\triangleq \exists \vec{v}_1\, \vec{v}_2\, \mathit{prot}_1\, \mathit{prot}_2.\ (c_1, c_2) \rightarrowtail (\vec{v}_1, \vec{v}_2) *\\
&\qquad \gamma_1 \mapsto_{\bullet} \mathit{prot}_1 * \gamma_2 \mapsto_{\bullet} \mathit{prot}_2 *\\
&\qquad \triangleright \begin{pmatrix} (\vec{v}_2 = \epsilon * \mathsf{interp}\ \vec{v}_1\ \mathit{prot}_1\ \mathit{prot}_2) \vee \\ (\vec{v}_1 = \epsilon * \mathsf{interp}\ \vec{v}_2\ \mathit{prot}_2\ \mathit{prot}_1) \end{pmatrix}\\
c \rightarrowtail \mathit{prot} &\triangleq \exists \gamma_1\, \gamma_2\, c_1\, c_2.\ \boxed{I\ \gamma_1\ \gamma_2\ c_1\ c_2} *\\
&\qquad ((c = c_1 * \gamma_1 \mapsto_{\circ} \mathit{prot}) \vee (c = c_2 * \gamma_2 \mapsto_{\circ} \mathit{prot}))
\end{aligned}$$

Setting the deeper technicalities related to the encoding in Iris aside, the most important parts are as follows. The two configurations informally described in the bullet list above are captured by the disjunction in the invariant I. The predicate $\mathsf{interp}\ \vec{v}\ \mathit{prot}_1\ \mathit{prot}_2$ captures that prot_2 contains zero or more receive (?) nodes that match up with the values $\vec{v}$ currently in the channel buffer. The clause $\Phi\ v\ (\lambda\, \mathit{prot}''.\ \mathsf{next}\ \mathit{prot}_2' = \mathit{prot}'')$ in the definition of interp describes the resources held by the messages in the channel buffer, while unifying the physical value v and the protocol tail prot_2' that is then used in the recursive call to interp.

In order to tie the proposition $c \rightarrowtail \mathit{prot}$ to the contents of the bidirectional channel c we follow the usual Iris methodology: we make use of Iris's invariants connective $\boxed{R}$ and a suitable "ghost theory". The invariant connective $\boxed{R}$ expresses that the proposition R holds at any time, *i.e.,* is invariant. As for the ghost theory we use higher-order ghost variables, whose rules are shown in Figure 12. Higher-order ghost variables come in pairs $\gamma \mapsto_{\bullet} \mathit{prot}$ and $\gamma \mapsto_{\circ} \mathit{prot}$, which should always hold the same protocol *prot*. They can be allocated 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 higher-order ghost variables is that they involve ownership of protocols of type iProto, which are defined in terms of Iris propositions iProp, which themselves are defined in terms of iProto. Due to this mutual recursion, the rule HO-ghost-agree only gives the equality between dependent separation protocols under a later modality.

5.4 Adequacy of Dependent Separation Protocols

With the above definitions at hand, we can now prove the rules Ht-newchan, Ht-send and Ht-recv from the combination of the logically atomic specifications in Section 5.1 and Iris's rules for invariants and ghost state. The proof of rule Ht-newchan is straightforward from HO-ghost-alloc, which is used to allocate two pairs of higher-order ghost variables for the chosen protocol and its dual. The crux of the proofs of Ht-send and Ht-recv is to update the higher-order ghost variables $\gamma \mapsto_{\bullet} \mathit{prot}$ and $\gamma \mapsto_{\circ} \mathit{prot}$ using HO-ghost-update to the tail of the protocol, and to transfer the resources described by the head of the protocol in and out of the invariant, when sending and receiving, respectively. This gives rise to the main result:

Theorem 5.1 (Adequacy of Actris). *Let ϕ be a first-order predicate over values and suppose the Hoare triple $\{\mathit{True}\}\ e\ \{\phi\}$ 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 [Jung et al. 2018b; Krebbers et al. 2017a]. Finally, note that safety implies *session fidelity*—any message that is received has in fact been sent.

6 COQ MECHANISATION

The definition of the Actris logic, its model, and the proofs of the examples in this paper have been fully mechanised using the Coq proof assistant [Coq Development Team 2019]. The mechanisation is built on top of the mechanisation of Iris [Jung et al. 2016, 2018b; Krebbers et al. 2017a] and the MoSeL Proof Mode (formerly Iris Proof Mode) [Krebbers et al. 2018, 2017b], which essentially provides an embedded proof assistant for separation logic in Coq. Building Actris on top of the Iris 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 Coq machinery, including the tactics for symbolic execution of non message-passing programs.
- Since Actris is essentially mechanised as an Iris library that provides support for the iProto type, the $c \rightarrowtail \mathit{prot}$ connective, the various operations on protocols, and the proof rules as lemmas, we get all present features of Iris for free.
- 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 give rise to a very small Coq development. The total size is about 3500 lines of code (comments and whitespace are included), of which 200 lines are used for the definitional semantics of channels, 1000 lines for the model, 450 lines for tactics, 1250 lines for the examples, and 600 lines for utilities (linked lists, permutations, *etc.*).

In order to implement Actris's tactics for symbolic execution, we followed the methodology described in the original Iris Proof Mode paper [Krebbers et al. 2017b], which means that the logic in Coq is presented in weakest precondition style rather than using Hoare triples. For handling `send` or `recv` we defined 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, and fixpoints so it can be written with a $\mathbf{!}\,\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\,prot$ or $\mathbf{?}\,\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\,prot$ construct in 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 $ are framed. In case the terms `(t1 .. tn)` are omitted, these are determined 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.

7 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 (Section 7.1) and process calculi (Section 7.2), session types (Section 7.3), endpoint sharing (Section 7.4), and verification efforts of map-reduce (Section 7.5).

7.1 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.

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 VST [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.

The work most closely related to ours, and the only work we know of that combines session types with separation logic, is by Tassarotti et al. [2017] whose main contribution is to 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. [2019] 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 address a different use case than we do.

7.2 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.

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 source code, but there is no formal connection between the specification logic and the underlying language.

Neither approach supports delegation or any concurrency paradigms other than message passing.

7.3 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]. Session types typically ensure that well-typed systems enjoy certain properties like session fidelity, progression, and deadlock freedom. Moreover, significant effort is placed into keeping type checking decidable.

Bocchi et al. [2010] push 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.

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 [Imai et al. 2019; Padovani 2017], 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 proving functional correctness of programs.

7.4 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 Section 3, sharing channels endpoints is often desirable.

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.

7.5 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.

8 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, and to determine if there is a translation, similar to the one from multi-party session types into binary session types, for such a version.

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 including 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], but it is not immediately obvious how to integrate that with Iris or Actris. 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.

Another direction for future work is to use dependent separation protocols for a logical relations model of session types. Similar to the RustBelt project [Jung et al. 2018a], this would give rise to an extensible approach for proving type soundness of type systems with session types, which can be used to establish that unsafe code in libraries has been safely encapsulated by its ADT.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their helpful feedback. We are grateful to Fabrizio Montesi, Daniel Gratzer, and the participants of the Iris workshop 2019 for discussions. Robbert Krebbers was supported by the Netherlands Organisation for Scientific Research (NWO), project number 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 (LNCS)*, Vol. 9600. 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*, Vol. 11423 LNCS. 611–639.

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).

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.

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

Coq Development Team. 2019. The Coq Proof Assistant Reference Manual, Version 8.9. (2019). https://coq.inria.fr/distrib/current/refman/

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).

Jafar Hamin and Bart Jacobs. 2019. Transferring Obligations Through Synchronizations. In *ECOOP*.

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. 2019. Coq Mechanization of Actris. Available online at https://gitlab.mpi-sws.org/iris/actris.

Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. 1998. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *proceedings of 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.

Bart Jacobs and Frank Piessens. 2011. Expressive Modular Fine-Grained Concurrency Specification. In *POPL*. 271–282.

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 (LNCS)*, Vol. 4137. 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. 2019. Aneris: A Mechanised Logic for Modular Reasoning about Distributed Systems. Submitted for publication.

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* 58, 1-2 (2004), 89–106.
Dimitris Mostrous and Vasco Thudichum Vasconcelos. 2014. Affine Sessions. In *COORDINATION*. 115–130.
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.
Peter W. O'Hearn. 2004. Resources, Concurrency and Local Reasoning. In *CONCUR*. 49–67.
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.
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.
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.