Sessions and Separation

Jonas Kastberg Hinrichsen
PhD Dissertation
IT University of Copenhagen

March 2021

Abstract

Session types and separation logic are two formalisms used to obtain strong guarantees about concurrent programs, each with their own benefits. Session types provide an intuitive protocol mechanism for specifying sequences of value exchanges, where each exchange is constrained to a specific type. In contrast, separation logic is often used for more involved specifications and proofs about shared-memory concurrency.

This thesis combines the two efforts by introducing *dependent separation protocols*—a session-type based reasoning mechanism in separation logic—and a *semantic session type system*—a session type system that combines the expressivity of separation logic with the simplicity of type checking.

Dependent separation protocols mimic the intuition behind session types, by describing sequences of value exchanges constrained by separation-logic propositions. The dependent separation protocols are formalised for a representative language with fine-grained concurrency, mutable state, and higher-order functions. This formalisation, which we call *Actris*, is built on top of the higher-order concurrent separation logic Iris. The use of Iris provides the ability to reason about recursion, substructural properties, and integration with existing concurrency models, such as shared-memory and lock-based concurrency. We demonstrate the expressivity of Actris by proving the correctness of several variants of a distributed merge sort algorithm, a distributed load-balancing mapper, and a variant of the map-reduce algorithm.

The semantic session type system is achieved using semantic typing and logical relations, where the types and judgements of the type system are defined in terms of Actris. The type system supports term- and session-type polymorphism, equi-recursion, and subtyping, as well as copyable types, shared and mutable reference types, and mutex types. The type system additionally supports manual typing proofs, where typing judgements that cannot be derived from the typing rules of the type system can instead be proven in the logic. This technique effectively lets us combine the full expressivity of the logic, with the simplicity of the syntax-directed type checking of the type system. We demonstrate this feature by manually proving the typing judgement of a program, where two threads operate on a single session-typed channel endpoint in parallel.

All of the results of the thesis have been mechanised in the interactive proof assistant Coq, on top of the existing mechanisation of Iris, along with tactic support for resolving proof obligations related to message passing in a style similar to working with session types.

Resumé

Sessionstyper og separationslogik er to metoder der benyttes til at opnå stærke garantier om samtidige programmer, med hver deres fordele. Sessionstyper giver en intuitiv protokolmekanisme til at specificere sekvesner af værdiudvekslinger, hvor hver udveksling er begrænset til en specifik type. Separationslogik er derimod ofte brugt i forbindelse med mere komplicerede specifikationer og beviser vedrørende delt hukommelse og samtidighed.

Denne afhandling kombinerer de to metoder ved at introducere *afhængige separationsprotokoller*—en sessionstype-baseret ræsonneringsmekanisme i separationslogik—og et *semantisk sessionstypesystem*—et sessionstypesystem der kombinerer ekspressiviteten af separationslogik med enkeltheden af type bestemmelse.

Afhængige separationsprotokoller efterligner intuitionen bag sessionstyper, da de beskriver sekvenser af værdiudvekslinger som er begrænset via separationslogikprædikater. De afhængige separationsprotokoller er formaliseret for et repræsentativt sprog med finkornet samtidighed, muterbar tilstand, og højere-ordens-funktioner. Denne formalisering, som vi kalder *Actris*, er bygget oven på den højere-ordens samtidige separationslogik Iris. Med brugen af Iris kan man ræsonnere om rekursion, substrukturelt ejerskab, og integration med eksisterende modeller om samtidighed, såsom delt hukommelse og låse-baseret samtidighed. Vi demonstrerer Actris ved at bevise korrektheden af flere varianter af en distribueret "merge sort"-algoritme, en distribueret belastnings-balanceret "mapper", og en variant af "map-reduce" algoritmen.

Det semantiske sessionstypesystem opnås via semantisk typning og logiske relationer, hvor typerne og typebedømmelsesmekanismen af typesystemet er defineret via Actris. Typesystemet understøtter term- og sessionstype polymorfi, rekursion, og subtyper, såvel som kopierbare typer, delte og muterbare referencetyper og mutextyper. Typesystemet understøtter derudover manuelle typningsbeviser, hvor typebedømmelser, der ikke kan udledes af type systemets typeregler, i stedet kan bevises i logikken. Denne metode lader os effektivt set kombinere logikkens fulde ekspressivitet med enkeltheden af typesystemets syntaksorienterede typebestemmelse. Vi demonstrerer denne funktionalitet ved manuelt at bevise typebedømmelsen af et program, hvor to tråde opererer på et enkelt sessionstypet kanal-slutpunkt i parallel.

Alle afhandlingens resultater er blevet mekaniseret i den interaktive bevisassistent Coq, oven på den eksisterende mekanisering af Iris, sammen med understøttelsen af taktikker til at løse bevisforpligtelser relateret til kommunikation, i en stil der minder om at arbejde med sessionstyper.

Acknowledgements

I would first and foremost like to thank Jesper Bengtson and Robbert Krebbers, for their tutelage as my supervisors, their collaboration as my co-authors, and their companionship as my friends. Of all the good memories that I have had during my time as a PhD student, I will treasure our intensive co-authoring sessions before a deadline the most, and our celebratory beer sessions only second to that.

I would then like to thank everyone at the IT University of Copenhagen, for creating a welcoming atmosphere. Especially so Frederik Madsen, my office mate for the longest time, with whom I shared the good times and endured the bad times. I would also like to thank the biking team, primarily spearheaded by Marco Carbone and Alessandro Bruni, who helped me achieve a better work life balance.

I extend my gratitude to the members of the committee, Derek Dreyer, Nobuko Yoshida, and Marco Carbone, for taking their time to review the thesis.

I would like to thank the hosts of my research stays abroad. First, Gregory Malecha of Bedrock Systems, who let me stay with his family until I found a temporary place to live in Boston, and to navigate the US infrastructure. A special thanks goes to my subsequent landlords Jordan and Nathan, and to my roommate Ben, who all made me feel at home. Secondly, Robbert Krebbers who hosted me twice at Delft University of Technology. Between chess matches, bouldering, and enjoying Belgian beer, it suffices to say that I had a great time staying in Delft, both during and outside of working hours. I can list many negative impacts that the corona pandemic had on my life, but cutting my time in Delft short is one of the worst offenders.

I would like to thank all of my friends. From the ones that I had before I started the degree, to the ones that I met during it, at the IT University of Copenhagen, the Oregon Programming Languages Summer School, the DeepSpec Summer School, and the Midlands Graduate School. Saying that I would not have been able to finish this thesis without them feels like an understatement.

A special thanks goes to Julie, who helped me through the most stressful time of my life, and who gave me the encouragement to seek the help that I needed.

Finally, I would like to thank my family. My sister and brother-in-law, who were the first to support my decision of pursuing a career in academia by "studying for three more years". Lastly, my parents, who always urged me to "do my best", while assuring that nothing more could be expected by anyone. These words always helped me push myself to improve my strengths, and occasionally to accept my weaknesses. I dedicate this thesis to them, my father Bjarne and my mother Susan.

Contents

Chapter 1

Introduction

The world is continuously moving towards concurrently executed programs, as we are reaching the physical limits of how fast a single chip can conduct computations. However, writing correct concurrent software is notoriously hard, and is therefore commonly approached using principled paradigms, such as message passing.

Even so, message-passing programs are prone to errors [Bagherzadeh et al. 2020], warranting increased correctness guarantees. A step in this direction is program testing, but as Dijkstra puts it "Program testing can be used to show the presence of bugs, but never to show their absence!". This is especially true for concurrent programs, that are often non-deterministic, where only one of many possible executions may lead to an error. We therefore seek to verify such message-passing programs by proving their *safety*, *i.e.,* no execution of the program has bad behaviour (such as calling a function with incorrect arguments), and *functional correctness*, *i.e.,* any execution of the program outputs the correct result (based on some specification).

A prominent approach to showing safety of message-passing programs is that of *session types*, a type system originally invented for process calculi [Honda 1993] that is still an active research topic today, featuring extensions such as shared channels [Balzer et al. 2019], dependent protocols [Thiemann and Vasconcelos 2020], and protocol weakening [Bravetti et al. 2021]. Orthogonally, showing safety and functional correctness of concurrent programs has been achieved with *concurrent separation logic* [O'Hearn 2004; Brookes 2004], a school of logics for reasoning about safe distribution of resources in a concurrent setting, from which descendants have been used to verify *e.g.,* C programs [Appel 2011] and Rust programs [Jung et al. 2018a].

In this thesis we combine session types and concurrent separation logic to establish a new reasoning mechanism, *dependent separation protocols*, for showing safety and functional correctness of programs that use message passing along with other concurrency models, such as lock-based synchronisation. We additionally combine the simplicity of type checking with the expressivity of logic proofs, by defining a *semantic type system* [Ahmed 2004] for session types. Finally, we mechanise all our results on top of the Iris framework [Iris Development Team 2021] in the Coq interactive proof assistant [Coq Development Team 2021], which includes the verification of a map-reduce algorithm, and a message-passing-based producer-consumer.

In this chapter we provide an overview of the field of session types and separation logic, and present the problem that the thesis seek to address. In particular, we first provide a brief overview of message passing (Section 1.1), safety and session types (Section 1.2), functional correctness and concurrent separation logic (Section 1.3), and an outline of some of the existing challenges in the field (Section 1.4). We then present the problem statement, the key idea to addressing the problem, and the contributions of the thesis (Section 1.5). Finally, we provide an insight to the contents of the included papers, and my role in conducting the work (Section 1.6).

1.1 Message Passing

Message passing is a concurrency paradigm in which threads are treated as actors in a network that synchronise via messages, rather than having shared access to mutable data [Hewitt et al. 1973]. The paradigm has been integrated as a first-class primitive of state-of-the-art languages, such as Go [The Go Team 2021] and Erlang [The Erlang Team 2021], while other languages, *e.g.,* Java and C#, support message passing through libraries [Akka 2021; Akka.NET 2021]. In such high-level languages messages are often guaranteed to arrive and be delivered in order. For this thesis we therefore consider binary message passing (where channels are between two parties) with these properties.

Message passing is often realised using a means of creating channels, sending messages and receiving messages, for which we use the following syntax:

$$\texttt{new\_chan}\ () \qquad \texttt{send}\ c\ v \qquad \texttt{recv}\ c$$

Here, `new_chan` () is used to allocate a new channel, creating two unbounded buffers $(\vec{v}_1, \vec{v}_2)$, while returning two *channel endpoints* c_1 and c_2. The operation `send` c_i v is used to send a value v over the channel, by enqueueing it in the corresponding buffer $\vec{v}_i$. Conversely `recv` c_i receives values by dequeueing them from the other buffer, *i.e.,* $\vec{v}_2$ if $i = 1$ and $\vec{v}_1$ if $i = 2$. The receive operation blocks until a value is available.

While the message-passing paradigm seek to avoid shared access to mutable data, it has been shown that developers often choose to combine the two [Tasharofi et al. 2013]. We therefore consider a representative ML-like language that supports mutable state, fork-based concurrency, locks, and the three message-passing primitives described above. The operational semantics, *i.e.,* the mathematical model, of the language is described in Section 2.1.2.

To illustrate the combination of message passing with shared memory, we consider the interaction between a pizza shop and a customer. To order pizzas the customer has to hand over their credit card (a reference to an integer), from which the pizza shop deducts the cost of the pizzas, after which they send back a receipt (a list of integers). This interaction is modelled by the following program:

```
order_pizza cc c :=
  send c [Margherita, Calzone];
  send c cc;
  let receipt := recv c in
  (sum receipt, !cc)
```

More precisely, the customer first sends the order of a margherita and a calzone pizza over the communication channel (`send` c [`Margherita`, `Calzone`]), and then send their credit card (`send` c cc). They then await a response from the pizza shop (`recv` c), expected to be the receipt (*receipt*). Finally the customer inspects (returns) the sum of the receipt (`sum` *receipt*) and the dereferenced value of the credit card (!cc).[1]

1.2 Safety and Session Types

We initially want to show safety of message-passing programs. However, safety is a general term informally described as "something [bad] will *not* happen" [Lamport 1977], and depends on the operational semantics of the language and what you want to prove about the programs. The safety that we consider is then whether the programs are *type-safe*, *e.g.,* that functions are applied to the right types of arguments, and that literal values are not used as functions. Safety of the `order_pizza` program presented in Section 1.1 then depends on a guarantee that the received message xs is a list of integers, as the operation `sum` xs can otherwise fail.

Showing safety is often achieved through type checking, *i.e.,* checking that the program is in compliance with the rules of a type system. Safety then follows under the condition that the type system satisfies *type safety*, *i.e.,* "[..] well-typed programs cannot 'go wrong'" as coined by Milner [1978]. A state-of-the-art type system for message passing is session types [Honda 1993], where channel endpoints are assigned a protocol. These protocols specify obligations to (primarily) send (**!**$A.\,S$) or receive (**?**$A.\,S$) a value of type A (*e.g.,* List **Z**) and then continue as S. Session types guarantee *session fidelity*, *i.e.,* that any received message has been sent, and that it has the denoted type. This can in turn be used to show that the received messages are used safely. The following session type captures the interaction with the pizza shop:

$$\mathit{pizza\_styp} \triangleq \mathbf{!}(\mathsf{List}\ \texttt{Pizza}).\,\mathbf{!}(\texttt{ref}\ \mathbf{Z}).\,\mathbf{?}(\mathsf{List}\ \mathbf{Z}).\,\mathit{pizza\_styp}$$

The session type specifically states that the customer must first send (**!**) the order, consisting of a list of pizzas (List `Pizza`), and their credit card (`ref` **Z**), after which they can receive (**?**) the receipt (List **Z**). The protocol then loops, awaiting a new order. The `order_pizza` program follows the protocol, as it first sends a list of pizzas ([`Margherita`, `Calzone`] : List `Pizza`), then the credit card reference (cc : `ref` **Z**), and then use the received value (*receipt* : List **Z**) in accordance with its type (`sum` *receipt*):

```
order_pizza cc c :=
    // c : !(List Pizza).!(ref Z).?(List Z). pizza_styp
  send c [Margherita, Calzone];
    // c : !(ref Z).?(List Z). pizza_styp
  send c cc;
    // c : ?(List Z). pizza_styp
  let receipt := recv c in
    // c : pizza_styp,    receipt : List Z
  (sum receipt, !cc)
```

[1] The hungry reader might notice that the ordered pizzas are never received. Sadly, pizzas cannot yet be sent digitally, and are thus not included in the protocol.

We can thus guarantee that the received receipt has the expected type, under the assumption that the pizza shop has similarly been type checked (thus complying with the protocol), and thereby guarantee that the program is type-safe.

However, the astute reader might have noticed that there is no guarantee that the credit card has not been charged more than the cost of the pizzas. To guarantee such a property we would instead need a proof of functional correctness.

1.3 Functional Correctness and Concurrent Separation Logic

Proving functional correctness can be achieved with Hoare logic [Hoare 1969], in which programs are ascribed with logical assertions, that specify propositions which hold before and after the execution of a program. These specifications are then proven by applying the rules of the logic, which capture how the assertions change throughout the evaluation of the program.

Extending the approach of Hoare logic to reason about stateful programs, *i.e.,* programs with mutable state, such as references, was later achieved with separation logic [Reynolds 2002; Ishtiaq and O'Hearn 2001]. Separation logic supports modular proofs of sub-programs, *i.e.,* proving the specifications of sub-programs in isolation. This is achieved by virtue of the separation property, which guarantees that the memory footprint of a sub-program is disjoint from the remaining program, thereby guaranteeing that they do not interfere with each other.

A later extension, concurrent separation logic [O'Hearn 2004; Brookes 2004], repurposed the separation property to implicitly capture the safe interleaving of concurrently executed threads, by guaranteeing that their executions are independent of each other. Concurrent separation logic additionally employ techniques for reasoning about the safe interleaving of threads with overlapping memory footprint. This direction has since spawned multiple descendants, for example VST [Appel 2011], FCSL [Nanevski et al. 2014], and Iris [Jung et al. 2015, 2018b]. Iris, in particular, is a higher-order concurrent separation logic, which supports the construction of high-level abstractions, such as locks and barriers, integrating them like first-class primitives, using higher-order ghost state [Jung et al. 2016].

Proving functional correctness of message-passing programs in the context of separation logic has been pursued in multiple efforts [Francalanza et al. 2011; Lozes and Villard 2012; Craciun et al. 2015; Oortwijn et al. 2016]. One particular effort is the work by Craciun et al. [2015], who combine separation logic with session types. They do so by replacing the message types with separation logic propositions, to specify additional details about the exchanged values, *e.g.,* to reason about exchanges of references and other channel endpoints.

The work by Craciun et al. [2015] is similar in spirit to prior work carried out by Bocchi et al. [2010]. They replace the message types with dependent first-order propositions, to specify additional details about the exchanged values, where the propositions can refer to previously exchanged values.

To prove that the program presented in Section 1.1 is correct, we use a technique similar to Bocchi et al. [2010] and Craciun et al. [2015], by extending session types with Iris propositions.

In particular, our protocols describe obligations to send ($!\,\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot}$) or receive ($?\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot}$) a value v constrained by a proposition P in the context of the logical binders $\vec{x}{:}\vec{\tau}$, after which the protocol continues as prot. We can then use the following protocol, to prove that the credit card is charged according to the receipt:

$$
\begin{array}{l}
\mathit{pizza\_prot} \triangleq \\
\quad !\,(v_1 : \mathsf{Val})(\vec{x}_1 : \mathsf{List}\ \mathtt{Pizza})\,\langle v_1\rangle\{\mathtt{is\_pizza\_list}\ v_1\ \vec{x}_1 * 0 < |\vec{x}_1|\}. \\
\quad !\,(\ell : \mathsf{Loc})(x : \mathbb{Z})\,\langle \ell\rangle\{\ell \mapsto x\}. \\
\quad ?(v_2 : \mathsf{Val}), (\vec{x}_2 : \mathsf{List}\ \mathbb{Z})\,\langle v_2\rangle\left\{\begin{array}{l}\mathtt{is\_int\_list}\ v_2\ \vec{x}_2 * |\vec{x}_1| = |\vec{x}_2| * \\ \ell \mapsto (x - (\mathtt{sum}\ \vec{x}_2))\end{array}\right\}. \\
\quad \mathit{pizza\_prot}
\end{array}
$$

The protocol states that the customer must first send a value v_1, which is an encoding of the list of pizzas $\vec{x}_1$, as captured by the proposition $\mathtt{is\_pizza\_list}\ v_1\ \vec{x}_1$. The side-condition $0 < |\vec{x}_1|$ captures that the list must be non-empty (to avoid prank calls). The side-condition holds alongside the list proposition, denoted by the separation conjunction ($*$), which for now can be thought of as regular propositional conjunction ($\wedge$). The customer must then send a location ℓ, representing their credit card, while noting that the original amount is x, captured by the separation logic points-to connective $\ell \mapsto x$. The pizza shop then returns the receipt v_2, which is a list of the cost of each pizza $\vec{x}_2$, captured by $\mathtt{is\_int\_list}\ v_2\ \vec{x}_2$. The side-condition $|\vec{x}_1| = |\vec{x}_2|$ states that the number of deductions must correspond to the number of ordered pizzas. Finally, the received proposition $\ell \mapsto (x - (\mathtt{sum}\ \vec{x}_2))$ captures that the new value stored in the credit card corresponds to the cost of the pizzas.

We can thus prove that the deducted amount is in accordance with the sum of the receipt, under the assumption that the pizza shop is in compliance with the protocol.

1.4 Challenges

This section presents some of the challenges related to reasoning about safety and functional correctness of message-passing programs. We first cover the importance of being able to reason about message passing in the context of other concurrency models, such as lock-based synchronisation (Section 1.4.1). We then illustrate the benefits of being able to combine the simplicity of safety checking via type systems, with the expressivity of interactive proofs in concurrent separation logic (Section 1.4.2). We finally discuss how mechanisation can be used to improve the reliability of type systems, logical systems, and the proofs carried out in them (Section 1.4.3).

1.4.1 Message Passing and other Concurrency Models

Message passing is often combined with mutable state and other concurrency models, such as lock-based synchronisation [Tasharofi et al. 2013]. To illustrate the benefits of combining message passing with locks consider two concurrent customers of the pizza shop, Jesper and Robbert. Jesper and Robbert are each having individual parties, and may want to order pizzas at some point during the evening, over the same channel endpoint. Sharing of channel endpoints is non-trivial as they are typically treated

as exclusive resources in the context of session types, meaning that they can only be accessed by one thread at a time. However, this exclusivity of channel endpoints is crucial to establish the guarantees of session types, as they rely on the fact that two threads do not operate on the same channel endpoint simultaneously.

To share channel endpoints between multiple threads we use locks, which guarantee safe shared access to a locked exclusive resource through mutual exclusion. In particular, a locked resource is guaranteed to remain unchanged outside of any critical section, *i.e.,* the sections where the shared exclusive resource is accessed. To access and momentarily modify the resources a threads must enter the critical section by acquiring the lock *lk*, using `acquire` *lk*, and afterwards release the lock, using `release` *lk*, to exit the critical section. The following program illustrates how the channel endpoint to the pizza shop is safely shared between both parties using a lock:

```
lock_example jesper_cc robbert_cc c :=
  let lk := new_lock () in
  ( // Jesper's party                    )  ||  ( // Robbert's party                    )
  ( acquire lk;                           )  ||  ( acquire lk;                           )
  (   send c [Margherita];                )  ||  (   send c [Calzone];                   )
  (   send c jesper_cc;                   )  ||  (   send c robbert_cc;                  )
  (   let jesper_receipt := recv c in     )  ||  (   let robbert_receipt := recv c in    )
  ( release lk;                           )  ||  ( release lk;                           )
  ( // Jesper's party cont.               )  ||  ( // Robbert's party cont.              )
  ( (sum jesper_receipt, ! jesper_cc)     )  ||  ( (sum robbert_receipt, ! robbert_cc)   )
```

The program executes the parties in parallel, which then race for the lock. Once the lock is acquired by either party, they can safely order pizzas, while the other party waits for the lock to be released.

Proving safety of programs which combine session types and lock-based synchronisation have been studied by Balzer and Pfenning [2017]; Balzer et al. [2019]. They add synchronisation primitives to their version of session types to guarantee mutual exclusion of channels during critical sections. However, this is different from having two separate mechanisms, one for locks and one for channels, that compose seamlessly. Additionally, their solution specifically address the combination of message passing and locks, and is not equipped to be extended with other concurrency models.

In regards to functional correctness, existing solutions for session-type based reasoning in separation logic [Craciun et al. 2015] do not integrate with other concurrency models, such as lock-based synchronisation. This is a result of building on top of more a traditional separation logic, as opposed to a logic which readily supports other concurrency models as first-class notions, *e.g.,* lock-based synchronisation, such as the higher-order concurrent separation logic Iris.

1.4.2 Safety via Functional Correctness

Proving safety of programs using type checking is often straightforward, and can in some cases even be done automatically using a decidable type checker. However, type checking is inherently not expressive enough to capture safety of some classes of racy yet safe programs, *i.e.,* programs which safely access exclusive resources concurrently.

Imagine that the pizza shop is having their annual pizza-palooza, in which a finite amount of people can get vouchers for free pizzas. Consider the following program and type, modelling two people concurrently requesting one of the last two pizzas:

$$(\lambda c.\, (\texttt{recv}\ c) \mid\mid (\texttt{recv}\ c)) : \texttt{chan}\ (\texttt{?Pizza.?Pizza.end}) \to \texttt{Pizza} \times \texttt{Pizza}$$

The program receives twice in parallel on the channel endpoint c, and returns the resulting pizzas as a pair. The program is safe, as any order of execution of the receive instructions results in a pair of some pizzas, as expected by the type annotation. However, as the channel endpoint is considered exclusive in conventional session type systems, they are incapable of showing safety of this program. This remains the case in the context of locks, since they would require the locked protocol to remain the same outside of the critical sections, which is not the case here, as each access strips one, and only one, receive step from the protocol, *i.e.*:[2]

$$\texttt{?Pizza.?Pizza.end} \rightsquigarrow \texttt{?Pizza.end} \qquad \text{or} \qquad \texttt{?Pizza.end} \rightsquigarrow \texttt{end}$$

As it is imperative that the pizza shop can safely continue the pizza-palooza tradition, we want to prove safety of the program, which is possible with concurrent separation logic, since its functional correctness proofs support reasoning about safe distributions of exclusive resources. However, proofs in a logic are almost always more time-consuming than type checking. We therefore wish to be able to type check the majority of a given program, and only prove safety of sub-programs in the logic when the expressivity of type checking is insufficient.

Achieving such a composition of type checking and manual typing proofs has been done by the RustBelt project [Jung et al. 2018a], which combines manual proofs of typing judgements for unsafe Rust code, with the syntactically typeable part of the language. They do so by employing the technique of semantic typing [Milner 1978; Ahmed 2004; Ahmed et al. 2010], in which types are given a logical interpretation, and the typing judgement is defined in a way that captures safety. Specifically, Jung et al. [2018a] define their type system in terms of the higher-order concurrent separation logic Iris, to capture resourceful types, such as Rust lifetimes.

1.4.3 Mechanisation

Proving properties such as safety and functional correctness using a *meta-theory*, *e.g.,* a session type system or a concurrent separation logic, is a means of increasing the reliability of software. However, this begs the question of how reliable these meta-theories are themselves. A meta-theory can similarly be prone to errors, as 1) proofs in the meta-theory might not actually guarantee the expected properties, and 2) proofs of individual programs in the meta-theory might have been carried out incorrectly.

Addressing these concerns can be done using mechanisation, where manual proofs are automatically checked by a machine, *e.g.,* via an interactive proof assistant such as Coq [Coq Development Team 2021], Isabelle/HOL [Nipkow et al. 2002], or

[2] The charitable reader may have noticed that the program could in fact be typed given a recursive type, however handing out infinite free pizzas is arguably a dubious business model.

Lean [de Moura et al. 2015]. Mechanisation has become popular over the years, and has been used for verification of cornerstone systems such as compilers [Leroy 2006; Kumar et al. 2014], operating systems [Klein et al. 2009; Gu et al. 2011], security protocols [Beringer et al. 2015], type systems [Jung et al. 2018a], and is even seeing use in industry [Bedrock Systems A/S 2021; The CertiK Team 2021].

We thus seek to mechanise the work carried out in this thesis, to strengthen the claim that our results are correct. In particular, we want to mechanise the meta-theory behind the proposed protocol mechanism, *i.e.,* that a proof in the meta-theory in fact guarantees functional correctness.

Doing so requires mechanising the operational semantics of the language, the meta-theory, and the connection between them. This mechanisation effort grows increasingly non-trivial for each feature of the meta-theory, *e.g.,* exclusive ownership, concurrency, and recursion. However, the mechanisation effort can be alleviated by building on top of an existing framework. Such an existing framework is the higher-order concurrent separation logic Iris in Coq [Iris Development Team 2021; Coq Development Team 2021]. Iris is parametric on an operational semantics, and includes a concrete instantiation with the operational semantics for an untyped functional language with higher-order functions, garbage collected higher-order mutable references, and fork-based concurrency, called HeapLang.

Additionally, we seek to carry out concrete proofs using the meta-theory, as evidence of its expressivity and usability, *i.e.,* that it is feasible to carry out proofs of larger and more realistic programs using the meta-theory.

To do so it is imperative to have a scalable method for carrying out proofs at the level of the meta-theory and its abstractions. This is the case as interactive proof assistants, such as Coq, are typically designed towards reasoning in their own logic. As a result, proofs in an embedded logic, such as Iris in Coq, often require manual handling of details that would be considered implicit in the embedded logic, *e.g.,* the distribution of substructural propositions in separation logic. A scalable method is the MoSeL framework (formerly called the Iris Proof Mode) [Krebbers et al. 2017b, 2018], which allows for reasoning at the level of concurrent separation logic in Coq, as well as being extensible with custom language- and abstraction-specific tactics.

Finally, following the trend of it becoming more common practice for researchers to mechanise their work, some mechanisation efforts have been conducted within the field of session types [Tassarotti et al. 2017; Thiemann 2019; Rouvoet et al. 2020; Gay et al. 2020; Castro et al. 2020]. However, there is a lack of mechanisations for more expressive session type systems, that combine multiple session type features such as recursion, polymorphism, and subtyping, along with types of other concurrent models such as mutex types for lock-based synchronisation.

1.5 Problem Statement and Contributions

This thesis concretely seeks to address the following three problems:

1. Session types provide an intuitive foundation for proving functional correctness of message-passing programs, however, existing session-type-based mechanisms do not compose with existing solutions for other concurrency models.

2. It is not possible to combine the expressivity of concurrent separation logic, *e.g.,* for proving safety of racy yet safe programs, with the simplicity of type checking in a session type system
3. It is becoming more common practice to mechanise new work, however, there is a lack of mechanisation efforts in the field of session types

The key idea to solving problem 1 is to draw inspiration from existing session-type based protocols for functional verification [Bocchi et al. 2010; Craciun et al. 2015]. We define a new protocol mechanism, that is both *dependent*, *i.e.,* protocol steps can rely on prior exchanges, and *resourceful*, *i.e.,* protocols can describe exchanges of exclusive resources, such as references or other channel endpoints. The protocols additionally support recursion, higher-order quantification, and weakening.

The protocol mechanism is formalised in Iris, using its support for constructing high-level abstractions, to integrate the protocol mechanism with Iris's expressive variant of concurrent separation logic, which include a first-class formalism of locks. To connect the protocol mechanism with an operational semantics, the message-passing primitives are then implemented on top of the HeapLang language, which has already been connected to the Iris meta-theory.

The key idea to solving problem 2 is to use the technique of *semantic typing*, drawing inspiration from the semantic type system for Rust in Iris by Jung et al. [2018a]. In particular, how we can integrate manual proofs of untypeable programs with the type system. Additionally, by building on top of Iris we inherit its affine properties for defining affine types, its recursion mechanisms for recursive types, and its lock library for mutex types. We use the proposed protocol mechanism as the semantic interpretation of session types with recursion, polymorphism, and subtyping.

Finally, the key idea to solving problem 3 is to mechanise all of our efforts on top of the Iris framework in Coq. In particular, we inherit the mechanisation of the operational semantics of HeapLang, and its connection to Iris's meta-theory. This lets us focus on the mechanisation of our protocol mechanism, the semantic type system, and the examples carried out in both of the systems.

Contributions The contributions made by this thesis are as follows:

1. We construct a first-class session-type based mechanism, *dependent separation protocols*, for proving functional correctness of message-passing programs in the higher-order concurrent separation logic Iris, using its support for high-level abstractions. The first-class mechanism integrates with Iris's existing concurrency mechanisms, such as shared mutable state and locks
2. We define a semantic type system for session types–based on the dependent separation protocols–along with shared- and mutable reference types, mutex types, term- and session subtyping, term- and session type polymorphism, recursive types, and support for manual proofs of "racy" yet safe programs
3. We fully mechanise both of the above results on top of the Iris mechanisation in Coq, with tooling support for proving typing judgements and functional specifications of concrete programs, along with various examples, including a functional proof of a map-reduce algorithm, and the manual typing proof of a message-passing-based producer-consumer

1.6 List of Publications and Manuscripts

Each of the contributions in Section 1.5 were achieved as a part of the work presented in the following collection of publications and manuscripts.

- Actris: Session-Type Based Reasoning in Separation Logic, ACM SIGPLAN Symposium on Principles of Programming Languages 2020 (POPL'20)
- Actris 2.0: Asynchronous Session-Type Based Reasoning in Separation Logic, Journal of Logical Methods in Computer Science (LMCS manuscript in review)
- Machine-Checked Semantic Session Typing, Certified Programs and Proofs 2021 (CPP'21) (recipient of Distinguished Paper Award)

More specifically, the first-class reasoning mechanism for message passing (contribution 1) is addressed by the POPL'20 paper and its LMCS journal version. The semantic session type system (contribution 2) is covered by the CPP'21 paper. Finally, the mechanisations (contribution 3) are included as artifacts of their respective papers. The LMCS manuscript and the CPP'21 paper have been reformatted and included in the thesis as Chapter 3 and Chapter 4, respectively. The remainder of this section is a brief overview of each paper (Sections 1.6.1 to 1.6.3), and concludes with references to the original papers and their mechanisation artifacts (Section 1.6.4).

1.6.1 Actris: Session-Type Based Reasoning in Separation Logic. Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers, POPL'20

This paper presents the first-class separation logic construct of *dependent separation protocols*, which can be thought of as a logical interpretation of session types.

In contrast to session types, which describe the exchanged values in terms of types, the dependent separation protocols describe exchanges using separation logic propositions, that can capture more precise properties than the types. An example of a dependent separation protocol, along with its corresponding session type, is the following, describing the protocol of a service that computes the length of a list:

Session type:	**Dependent separation protocol:**
$?(\mathsf{List}\ \mathbf{Z}).\,!\mathbf{Z}.\,\mathtt{end}$	$?(v : \mathsf{Val}), (\vec{x} : \mathsf{List}\ \mathbb{Z})\,\langle v\rangle\{\mathtt{is\_int\_list}\ v\ \vec{x}\}.\,!\,\langle\lvert\vec{x}\rvert\rangle.\,\mathbf{end}$

The dependent separation protocol states that the channel first receives a value v, which is an encoding of a list of integers $\vec{x}$, as determined by $\mathtt{is\_int\_list}\ v\ \vec{x}$. It then sends back the length of the list $|\vec{x}|$, with no propositional restriction.

The protocol construct has been integrated with the Iris concurrent separation logic framework, allowing it to *e.g.,* carry exclusive ownership of references and other channel endpoints. Additionally, the integration allows combining the protocols with lock-based reasoning, which is demonstrated in the paper, along with a case study in which we verify a map-reduce algorithm.

Personal Contribution The publication is joint work, with close collaboration between all co-authors on all sections. As first author I spearheaded the writing of all

of the sections except the related work section (initial version by Jesper Bengtson), and the model and mechanisation sections (initial version by Robbert Krebbers).

The publication is accompanied by an artifact consisting of the mechanisation of the dependent separation protocols, the channel implementation, and the examples of the paper. The mechanisation was developed in collaboration with Robbert Krebbers. Krebbers particularly carried out the implementation of the recursive domain equation of the dependent separation protocols, and the proof of the load-balancing mapper and map-reduce examples. I did the rest under his supervision, *e.g.,* carried out the protocol definition, and connected it to the channel implementation.

1.6.2 Actris 2.0: Asynchronous Session-Type Based Reasoning in Separation Logic. Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers, LMCS manuscript in review

This paper extends the dependent separation protocol mechanism with a notion similar to asynchronous session subtyping, where protocols can be weakened to allow the composition of more programs that communicate. Subtyping achieved this by permitting (1) sending more, (2) receiving less, and (3) eagerly sending before receiving. In addition to conventional subtyping, our subprotocol mechanism is resourceful. It allows "framing" resources that are sent in excess, which can then be re-obtained during a subsequent receive. The mechanism also enjoys the full expressivity of Iris's step-indexed recursion, allowing weakening under the recursive operator. An example of a subprotocol, along with its corresponding session type subtyping, is the following which captures how the list length service may disregard the type of the lists elements:

Subtype:	**Subprotocol:**
$\mathbf{?}(\mathsf{List}\ \mathbf{Z}).\,\mathbf{!}\mathbf{Z}.\,\mathtt{end}$	$\mathbf{?}(v : \mathsf{Val}), (\vec{x} : \mathsf{List}\ \mathbb{Z})\ \langle v\rangle\{\mathtt{is\_int\_list}\ v\ \vec{x}\}.\,\mathbf{!}\ \langle\lvert\vec{x}\rvert\rangle.\,\mathbf{end}$
$<:\ \mathbf{?}(\mathsf{List\ any}).\,\mathbf{!}\mathbf{Z}.\,\mathtt{end}$	$\sqsubseteq\ \mathbf{?}(v : \mathsf{Val}), (\vec{w} : \mathsf{List\ Val})\ \langle v\rangle\{\mathtt{is\_list}\ v\ \vec{w}\}.\,\mathbf{!}\ \langle\lvert\vec{w}\rvert\rangle.\,\mathbf{end}$

This subprotocol relation captures that the channel endpoint can receive a list of arbitrary values $\vec{w}$, instead of a list of integers $\vec{x}$. It does so formally by weakening the received proposition $\mathtt{is\_int\_list}\ v\ \vec{x}$, and its associated binder $\vec{x}$, to the strictly weaker proposition $\mathtt{is\_list}\ v\ \vec{w}$, along with its binder $\vec{w}$. The tail of the protocol is suitably updated, to send back the length of the list of arbitrary values $|\vec{w}|$.

The manuscript additionally extends the model and mechanisation sections of the POPL'20 publication, to provide further insight into how the Actris framework is defined on top of Iris, and how the mechanisation made use of the Iris Proof Mode.

Personal Contribution The manuscript is joint work where I wrote the initial draft of each of the three new/revised sections, while my co-authors provided feedback and revisions. The extension of the mechanisation was achieved as a collaboration with Robbert Krebbers, where Krebbers solved technically challenging problems related updating the recursive domain equation, the step-indexing of Iris, and the tactic support for the subprotocol mechanism, while I worked on the rest under his supervision, such as conceptualising the subprotocol definition and integrating it with our existing protocol framework.

1.6.3 Machine-Checked Semantic Session Typing. Jonas Kastberg Hinrichsen, Daniël Louwrink, Jesper Bengtson, and Robbert Krebbers, CPP'21, recipient of Distinguished Paper Award

This paper draws a formal connection between the dependent separation protocols and session types, by defining and mechanising a semantic session type system as a logical relation on top of Actris.

In doing so, we obtain a rich session type system, with term- and session type subtyping, polymorphism, and equi-recursion, in addition to copyable types, mutable and shared reference types, and mutex types. Furthermore, it integrates the expressiveness of the underlying logic with the simplicity of the session type system, by enabling manually proving typing judgements, of uncheckable programs, such as the one originally presented in Section 1.4.2:

$$\lambda c.\, (\texttt{recv}\ c) \mid\mid (\texttt{recv}\ c) : \texttt{chan}\ (\texttt{?Pizza}.\, \texttt{?Pizza}.\, \texttt{end}) \to \texttt{Pizza} \times \texttt{Pizza}$$

As previously stated, the program is safe, as the order in which it receives does not matter, but even so, it cannot be type checked by conventional session type systems, as the channel type cannot be distributed to both parallel threads. However, we can manually prove the typing judgement, using the underlying logic, and then use it in the context of type checking for compound programs.

Personal Contribution The writing was joint work with close collaboration between the co-authors. I wrote the initial draft of all sections excluding the mechanisation section (initial version by Robbert Krebbers) and the related work section (initial version by Jesper Bengtson). The mechanisation was developed in collaboration with Daniël Louwrink, who laid the ground work inspired by the RustBelt project, and Robbert Krebbers, who assisted with technical Coq challenges *e.g.,* related to applying polymorphic typing rules. I did the rest, such as extending the system with subtyping and carrying out the mechanisation of the examples in the paper.

1.6.4 Mechanisations and Artifacts

The Coq mechanisation for the three paper share the same code base and thus repository, for which the newest version is found at https://gitlab.mpi-sws.org/iris/actris. Additionally, each of the published papers have an associated self-contained persistently **archived** virtual machine, along with a **repo branch** of the repository reflecting the source code at their time of publication:

- POPL'20: https://dl.acm.org/doi/10.1145/3371074
  - archive: https://dl.acm.org/do/10.1145/3373096/full/
  - repo branch: https://gitlab.mpi-sws.org/iris/actris/-/tree/popl20
- LMCS: https://iris-project.org/pdfs/2020-actris2-submission.pdf (pre-print)
  - repo branch: https://gitlab.mpi-sws.org/iris/actris/-/tree/lmcs
- CPP'21: https://doi.org/10.1145/3437992.3439914
  - archive: https://zenodo.org/record/4322752
  - repo branch: https://gitlab.mpi-sws.org/iris/actris/-/tree/cpp21

Chapter 2

Background

The work of this thesis is carried out on top of the higher-order concurrent separation logic Iris in Coq [Jung et al. 2015, 2018b; Iris Development Team 2021]. Iris is language independent, *i.e.,* it is parametric on an operational semantics. For the purpose of this thesis we consider the instantiation with an untyped functional language with higher-order functions, garbage collected higher-order mutable references, and fork-based concurrency, called HeapLang.

In this chapter we provide a technical background for the remaining chapters. In particular, we first present the operational semantics of HeapLang, along with existing implementations of parallel composition and locks, as well as our own implementations of linked lists and binary channels for message passing (Section 2.1). We specifically consider the problem of capturing all of the possible interleavings of a concurrent program, and how this is modelled semantically.

We then provide a precise definition of the safety and functional correctness properties, based on the operational semantics of HeapLang, that we consider (Section 2.2). We additionally show how these properties can be established using a type system or Hoare logic, and how the benefits of these techniques can be combined using the method of semantic typing. In doing so, we precisely capture the functional correctness property that we obtain with the Actris framework presented in Chapter 3, and the safety property that we obtain with the semantic type system built on top of Actris presented in Chapter 4.

We finally give an overview of the Iris framework, and demonstrate how it can be used to prove specifications of concurrent programs that share exclusive resources using locks (Section 2.3). That is to give an intuition for how we may later use Iris as the foundation for constructing our session-type based protocols, and for how we prove the specifications of our binary channels for message passing in the Actris framework in Chapter 3. We additionally present some of the features of Iris that we use to model various types in our semantic session type system presented in Chapter 4.

The remaining chapters of the thesis are self-contained in regards to session types, and thus we do not cover them in this chapter.

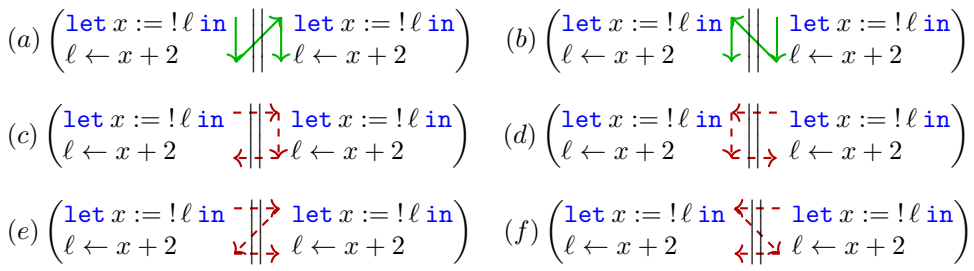

Figure 2.1: The possible interleavings of `interleaving_example`. The interleavings (a) and (b) yield the correct result, in contrast to (c), (d), (e), and (f) which yield an incorrect result. Arrows indicate the flow of time.

2.1 Operational Semantics

In this section we cover the operational semantics of HeapLang and the abstractions, that we use in the remaining chapters, built on top of it. In particular, we first give a brief account of the intricacies associated with concurrent operational semantics (Section 2.1.1). We then present the concrete operational semantics of the HeapLang language (Section 2.1.2). We then cover the existing implementations of locks and parallel composition (Section 2.1.3). Finally, we cover our own implementations of mutable linked lists (Section 2.1.4), and bidirectional channels for message passing (Section 2.1.5).[1]

2.1.1 Interleavings in Concurrent Operational Semantics

Reasoning about concurrent software is notoriously hard, since the program instructions of individual threads are sporadically interleaved. As a consequence of this, even though some possible execution orders might produce the intended result, there can still exist interleavings that result in faulty behaviour. The following program illustrates this problem:

$$\texttt{interleaving\_example}\ \ell := \left(\begin{array}{l}\texttt{let}\ x := !\ell\ \texttt{in} \\ \ell \leftarrow x + 2\end{array}\right) \Vert \left(\begin{array}{l}\texttt{let}\ x := !\ell\ \texttt{in} \\ \ell \leftarrow x + 2\end{array}\right)$$

The program concurrently updates a value x stored in the reference ℓ twice. It does so by executing two threads in parallel, that reads the value from the reference, adds 2 to the value, and stores the updated value back in the reference.

The problem is that four of its six possible interleavings result in an incorrect result, as shown in Figure 2.1.[2] The program correctly adds 4 to the referenced value

[1] The content of (Section 2.1.5) is primarily replicated from the corresponding section on channel semantics in (Section 3.6.5), in order to give a complete overview of the language semantics.

[2] We assume that the language has a sequentially-consistent memory model.

in the first two cases (a) and (b), as the reference is both read and updated by one thread, before the other thread reads the updated value. However, in the remaining cases (c), (d), (e), and (f), both threads read the original value (x), update their local view of the value (to $x + 2$), and then store that in the reference. As a result only one update is effectively applied, resulting in the reference now containing $x + 2$, as opposed to $x + 4$. We fix this bug by protecting the critical sections with a lock in Section 2.1.3, and prove that the program is both safe and correct in Section 2.3.

To obtain proper guarantees about our concurrent programs, we want our proofs of safety and functional correctness to account for any possible interleaving. This is achieved by reflecting the interleaving behaviour in the operational semantics.

2.1.2 The Semantics of HeapLang

HeapLang is an untyped functional language with higher-order functions, garbage collected higher-order mutable references, and fork-based concurrency, for which the syntax is as follows:[3]

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

We omit the the standard unary and binary operations such as equality, addition, and subtraction. The language includes the standard operations for heap manipulation. New references are created using $\texttt{ref}\ e$, dereferenced using $!\, e$, and assigned to using $e_1 \leftarrow e_2$. Concurrency is supported via $\texttt{fork}\ \{e\}$, which spawns a new thread, that executes the expression e in the background. The language also supports the atomic compare-and-set ($\texttt{CAS}$) operation, which is used to implement the lock in Section 2.1.3.

Derived Notions We derive various notions as syntactic sugar, *i.e.,* definitions in the meta-language by use of $\triangleq$, such as lambda expressions ($\lambda x.\, e$), let-bindings ($\texttt{let}\ x := e_1\ \texttt{in}\ e_2$), and options ($\texttt{None}, \texttt{Some}\ e$), from the core syntax:

$$
\begin{aligned}
\lambda x.\, e &\triangleq \texttt{rec}\ \_\ x := e & \texttt{None} &\triangleq \texttt{inj}_1\ () \\
\texttt{let}\ x := e_1\ \texttt{in}\ e_2 &\triangleq (\lambda x.\, e_2)\ e_1 & \texttt{Some}\ e &\triangleq \texttt{inj}_2\ e \\
e_1;\ e_2 &\triangleq \texttt{let}\ \_ := e_1\ \texttt{in}\ e_2 & \texttt{skip} &\triangleq (\lambda \_.\, ())\ ()
\end{aligned}
$$

We often write definitions as $f\ x_1 \cdots\ x_n := e$ rather than $f \triangleq \texttt{rec}\ f\ x_1 \cdots\ x_n := e$.

[3] HeapLang has more features, *e.g.,* arrays, but for brevity's sake we describe a more concise version of the operational semantics. However, the mechanisation effort of the thesis is built on top of the full version of HeapLang.

$$
\begin{array}{rcll}
((\texttt{rec}\ f\ x := e)(v); \sigma) & \to_{\mathsf{h}} & (e[v/x][(\texttt{rec}\ f\ x := e)/f]; \sigma; [\,]) & \\
(\texttt{fst}\ (v_1, v_2); \sigma) & \to_{\mathsf{h}} & (v_1; \sigma; [\,]) & \\
(\texttt{snd}\ (v_1, v_2); \sigma) & \to_{\mathsf{h}} & (v_2; \sigma; [\,]) & \\
(\texttt{if true then}\ e_1\ \texttt{else}\ e_2; \sigma) & \to_{\mathsf{h}} & (e_1; \sigma; [\,]) & \\
(\texttt{if false then}\ e_1\ \texttt{else}\ e_2; \sigma) & \to_{\mathsf{h}} & (e_2; \sigma; [\,]) & \\
\left(\begin{array}{l}\texttt{match}\ (\texttt{inj}_i\ v)\ \texttt{with} \\ \quad (\texttt{inj}_1\ x) \Rightarrow e_1 \\ \mid (\texttt{inj}_2\ x) \Rightarrow e_2 \\ \texttt{end}\end{array} ; \sigma\right) & \to_{\mathsf{h}} & (e_i[v/x]; \sigma; [\,]) & \text{if } i \in \{1, 2\} \\
(\texttt{ref}\ v; \sigma) & \to_{\mathsf{h}} & (\ell; \sigma[\ell \leftarrow v]; [\,]) & \text{if } \sigma(\ell) = \bot \\
(!\,\ell; \sigma[\ell \leftarrow v]) & \to_{\mathsf{h}} & (v; \sigma[\ell \leftarrow v]; [\,]) & \\
(\ell \leftarrow w; \sigma[\ell \leftarrow v]) & \to_{\mathsf{h}} & ((); \sigma[\ell \leftarrow w]; [\,]) & \\
(\texttt{fork}\ \{e\}; \sigma) & \to_{\mathsf{h}} & ((); \sigma; [e]) & \\
(\texttt{CAS}\ \ell\ w_1\ w_2; \sigma[\ell \leftarrow v]) & \to_{\mathsf{h}} & (\texttt{true}; \sigma[\ell \leftarrow w_2]; [\,]) & \text{if } v = w_1 \\
(\texttt{CAS}\ \ell\ w_1\ w_2; \sigma[\ell \leftarrow v]) & \to_{\mathsf{h}} & (\texttt{false}; \sigma[\ell \leftarrow v]; [\,]) & \text{if } v \neq w_1
\end{array}
$$

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

Figure 2.2: The operational semantics of HeapLang.

Operational Semantics The small-step operational semantics of HeapLang can be found in Figure 2.2. It consists of three layers of reductions:

- *term reductions*, $(\_\,;\_ \to_{\mathsf{h}} \_\,;\_\,;\_) \in (\mathsf{Expr} \times \mathsf{State}) \times ((\mathsf{Expr} \times \mathsf{State}) \times \mathsf{List\ Expr})$ which capture how individual operations reduce
- *thread-local reduction*, $(\_\,;\_ \to_{\mathsf{t}} \_\,;\_\,;\_) \in (\mathsf{Expr} \times \mathsf{State}) \times ((\mathsf{Expr} \times \mathsf{State}) \times \mathsf{List\ Expr})$ which capture the evaluation order of sub-expressions
- *threadpool reduction*, $(\_\,;\_ \to_{\mathsf{tp}} \_\,;\_) \in (\mathsf{List\ Expr} \times \mathsf{State}) \times (\mathsf{List\ Expr} \times \mathsf{State})$ which capture the concurrent semantics of the language

Here $\sigma \in \mathsf{State} \triangleq \mathsf{Loc} \xrightarrow{\mathsf{fin}} \mathsf{Val}$, *i.e.*, a finite partial map from allocated locations to their stored values.

The *term reduction* $\to_{\mathsf{h}}$ captures how the language primitives reduce. The reduction is defined as a relation from a configuration $(\mathsf{Expr} \times \mathsf{State})$, to a configuration and a range of newly spawned threads $((\mathsf{Expr} \times \mathsf{State}) \times \mathsf{List\ Expr})$. The individual primitive reductions are mostly standard, and are shown in Figure 2.2. The reduction of the fork-expression $(\texttt{fork}\ \{e\}; \sigma) \to_{\mathsf{h}} ((); \sigma; [e])$ captures how a new thread e is spawned by adding it to the list of newly spawned threads $[e]$.

The *thread-local reduction* $\rightarrow_{\mathsf{t}}$ is used to capture the deterministic evaluation order of sub-expressions, using the following *evaluation context* syntax:

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

In particular, expressions are represented in the form $K[\,e\,]$, capturing that e is the head-expression of a context K. HeapLang reduces from right-to-left, meaning that in expressions such as $e_1 \leftarrow e_2$ the expression e_2 reduces before e_1. This is determined by its corresponding context syntax, from which we initially get $(e_1 \leftarrow \bullet)[\,e_2\,]$. If e_2 reduces to a value v_2 the context syntax dictates that the hole moves to e_1 yielding $(\bullet \leftarrow v_2)[\,e_1\,]$. If e_1 reduces to a value v_1 we finally end up with $v_1 \leftarrow v_2$, as there is no context syntax where both constituents are values.

Finally, the *threadpool reduction* $\rightarrow_{\mathsf{tp}}$ captures the concurrent semantics of HeapLang, being the top-level reduction of the language. The threadpool reduction is defined as a relation over all concurrently running threads and the heap state ($\mathsf{List\ Expr} \times \mathsf{State}$) to a new set of threads and an updated heap state ($\mathsf{List\ Expr} \times \mathsf{State}$). At each step a thread is picked arbitrarily and reduced once via the thread reduction $\rightarrow_{\mathsf{t}}$, applying the updates to the heap state, while adding any forked-off threads $\vec{e}$ to the threadpool $T \cdot [e_2] \cdot T' \cdot \vec{e}$.

As the reduced thread is picked arbitrarily, a program execution simulation can result in any possible interleaving, even ones where the same thread is infinitely picked repeatedly (assuming it loops), effectively accounting for an unfair scheduler. However, as will be detailed in Section 2.2 we are not concerned with termination, and thus an infinitely executing program is deemed well-behaved. As a result of the arbitrary reduction of threads, any proof over a program execution must consider every possible interleaving, even the unfair ones.

2.1.3 Implementation of Locks and Parallel Composition

Iris's HeapLang library [Iris Development Team 2021, HeapLang] includes implementations of locks and the parallel composition, found in Figure 2.3.

The locks are encoded as a boolean reference, where `false` indicates that the lock is free, and `true` indicates that the lock is taken. New locks are constructed with `new_lock` (), which does so by allocating a reference to `false`, *i.e.,* a free lock. Acquiring the lock is primarily done with try_acquire *lk*, which uses a compare-and-set (`CAS`) operation to atomically query whether the lock is free (`false`). If the lock is free, it is taken by atomically setting the reference to `true`. The function returns whether the `CAS` operation was successful. To make lock acquisition a *blocking operation*, *i.e.,* one that only continues once it succeeds, we implement the actual acquire function `acquire` *lk* by wrapping try_acquire *lk* in a loop, which only terminates once the `CAS` operation succeeds. Finally, the lock can be released with `release` *lk* which sets the reference back to `false`.

Implementation of locks:

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

Implementation of parallel composition:

$$
\begin{aligned}
&\texttt{spawn}\ f := \texttt{let}\ cb := \texttt{ref None in}\\
&\qquad\qquad \texttt{fork}\ \{cb \leftarrow \texttt{Some}\ (f\ ())\};\\
&\qquad\qquad cb\\
&\texttt{join}\ cb := \texttt{match}\,!\,cb\ \texttt{with}\\
&\qquad\qquad \texttt{None} \Rightarrow \texttt{join}\ cb\\
&\qquad\qquad \mid \texttt{Some}\ x \Rightarrow x\\
&\qquad\qquad \texttt{end}
\end{aligned}
\qquad\qquad
\begin{aligned}
&(e_1 \mid\mid e_2) \triangleq \texttt{let}\ h := \texttt{spawn}\ (\lambda\_.\, e_1)\ \texttt{in}\\
&\qquad\qquad \texttt{let}\ v_2 := e_2\ \texttt{in}\\
&\qquad\qquad \texttt{let}\ v_1 := \texttt{join}\ h\ \texttt{in}\\
&\qquad\qquad (v_1, v_2)
\end{aligned}
$$

Figure 2.3: Implementation of locks and parallel composition on top of HeapLang.

The parallel composition $(e_1 \mid\mid e_2)$ computes the results of e_1 and e_2 in parallel, and returns the resulting values v_1 and v_2 as a tuple. The implementation depends on two auxiliary functions `spawn` and `join`. The `spawn` function takes a thunk, *i.e.*, a computation delayed by wrapping it in a closure, which it computes in a forked-off thread, while returning a callback cb. The callback is a reference to an option, where `None` indicates that the value has yet to be computed, and `Some` x means that the result x is ready. The `join` function takes a callback cb, and checks if the result is ready. If it is not, the function loops. If the result is ready, it is returned. The parallel composition $(e_1 \mid\mid e_2)$ is defined in the meta-language (using $\triangleq$), as opposed to using a lambda abstraction in HeapLang, to avoid eagerly computing e_1 and e_2, when passing them to the function. The function spawns a new thread for computing the result of e_1 using `spawn` $(\lambda\_.\, e_1)$, after which it computes the result of e_2 locally. When the evaluation of e_2 is done, the function awaits the result of e_1 using `join`. Finally, the results v_1 and v_2 are returned as a tuple.

The Lock-Protected Interleaving Example By employing the locks we can define a correct variant of the `interleaving_example` example from Section 2.1.1:

$$
\begin{aligned}
&\texttt{locked\_interleaving\_example}\ \ell :=\\
&\quad \texttt{let}\ lk := \texttt{new\_lock}\ ()\ \texttt{in}\\
&\quad \begin{pmatrix}
\texttt{acquire}\ lk; & \texttt{acquire}\ lk;\\
\quad \texttt{let}\ x := !\,\ell\ \texttt{in} & \quad \texttt{let}\ x := !\,\ell\ \texttt{in}\\
\quad \ell \leftarrow x + 2; & \quad \ell \leftarrow x + 2;\\
\texttt{release}\ lk & \texttt{release}\ lk
\end{pmatrix};\\
&\quad \texttt{acquire}\ lk
\end{aligned}
$$

```
  lnil () := ref None
lcons v l := l ← Some (v, ref (!l))
lisnil l  :=  match !l with
               None   ⇒ true
              | Some p ⇒ false
              end
```

```
lsnoc l v := match !l with
               None   ⇒ l ← Some (v, ref None)
              | Some p ⇒ lsnoc (snd p) v
              end
    lpop l := match !l with
               None   ⇒ () ()
              | Some p ⇒ l ← !(snd p); fst p
              end
```

Figure 2.4: A selection of the definitions for the implementation of linked lists on top of HeapLang. We omit the functions `llength`, `lappend`, `lreverse`, `lmerge`, and `lsplit`, used in the remaining chapters.

The lock guarantees that only one thread can access the location ℓ at a time. As a result, the possible interleavings of the program reduce to the correct cases (a) and (b), thus guaranteeing that the program correctly adds 4 to the stored value x.

We acquire the lock after the parallel composition, to simulate a deallocation of the lock. This is done to obtain a simpler proof of the functional correctness of the program in Section 2.3, inspired by [Jourdan and Krebbers 2018, Exercise 3].

2.1.4 Implementation of a Mutable Linked List

We implement mutable linked lists on top of HeapLang, for which a selection of definitions can be found in Figure 2.4. We take the common approach to linked lists, encoding them as references to options. A list may then either be empty (`None`), or a cell (`Some` (v, l)), where v is the value stored in the cell and l is a reference to the next cell. This is made apparent by the definition of `lnil` (), which allocates a reference to a new empty list `ref None`. Consequentially, the definition `lcons` v l extends a linked list l by letting the reference point to a new cell with the new value v, and a newly allocated reference to the original tail `ref` $(!l)$.

The linked list implementation includes a small library of functions of which a selection is shown in Figure 2.4. In particular, we use the functions `lisnil`, `lsnoc`, and `lpop`, for the implementation of the binary channels for message passing in Section 2.1.5. The `lisnil` l function queries whether the list l is empty, by matching on its option constructor. The `lsnoc` v l function adds a value v to the end of the list l, by traversing the entire list. Finally, the `lpop` l function removes the first element of the list l and returns it. The function fails if the list is not empty, which is captured using the illegal function application () (), *i.e.,* an application of the unit value () to itself, as if it were a function. As there are no valid reductions for a function application on values that are not functions, the program gets stuck, which will be discussed further in Section 2.2. Finally, note that we often use $|l|$ instead of `llength` l.

```
new_chan () := let (l, r, lk) := (lnil (), lnil (), new_lock ()) in
               ((l, r, lk), (r, l, lk))
   send c v := let (l, r, lk) := c in
               acquire lk;
                 lsnoc l v; skipN |r|;
               release lk
try_recv c := let (l, r, lk) := c in
               acquire lk;
                 let ret := (if (lisnil r) then (None) else (Some (lpop l))) in
               release lk; ret
    recv c := match (try_recv c) with
                None   ⇒ recv c
              | Some v ⇒ v
              end
```

Figure 2.5: Implementation of binary channels for message passing in HeapLang.

2.1.5 Implementation of Binary Channels for Message Passing

We implement our channels on top of HeapLang, using a pair of mutable linked lists protected by a lock, as shown in Figure 2.5. Each linked list represent a buffer of messages sent from one endpoint to the other.

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

Values are sent over a channel endpoint (l, r, lk) using the `send` function. This function operates in an atomic fashion by first acquiring the lock via `acquire` lk, thereby entering the critical section, after which the value is enqueued (*i.e.,* appended to the end) of the endpoint's own buffer using `lsnoc` l v. The `skipN` $|r|$ instruction is a no-op that is inserted to aid the proof. We explain the reason why this instruction is needed in Section 3.6.6.

Values are received over a channel endpoint (l, r, lk) using the `recv` function. The function performs a loop that repeatedly calls the helper function `try_recv`. It loops whenever `try_recv` fails (*i.e.,* returns `None`), and terminates when it succeeds (*i.e.,* returns `Some` v), returning the value v. The helper function `try_recv` attempts to receive a value atomically, and fails if there is no value in the other endpoint's buffer. It does so by first acquiring the lock lk with `acquire` lk, after which it checks whether the other endpoint's buffer is empty using `lisnil` r. If the buffer is empty nothing is returned (*i.e.,* `None`). If it is non-empty the value is dequeued and returned (*i.e.,* `Some (lpop` l`)`). Finally, the lock lk is released with `release` lk.

2.2 Safety, Functional Correctness, and Semantic Typing

This section elaborates on how safety and functional correctness of a program can be proven using type checking and Hoare logic. First, we formally define what we mean by safety, and how it can be shown via type checking (Section 2.2.1). In doing so, we illustrate a key limitation of conventional type systems, in that they cannot type check programs whose safety rely on runtime behaviour, *e.g.,* programs with unsafe branches that are never executed. We then formally define what we mean by functional correctness, how it can be shown through proofs in a program logic, and how it can be used to prove safety of programs beyond the scope of type checking (Section 2.2.2). Finally, we cover the technique of semantic typing, and how it can be used to combine the benefits of type checking and interactive proofs (Section 2.2.3).

2.2.1 Safety

Safety of a program has informally been described as "something [bad] will *not* happen" [Lamport 1977]. We interpret this as a program always being able to reduce, *i.e.,* being able to take a step according to the small-step operational semantics. This implies that the program is type-safe, as illegal (primitive) function applications cannot reduce. An example of an illegal function application is ()(), as the literal unit value () is used as a function. Additionally, the safety interpretation implies that programs are memory-safe, as the reduction rule for dereferencing requires that the location is in the heap to take a step. We capture the safety property as follows:

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

The property states that an expression e is safe, whenever any expression $e' \in T$ that it reduces to is either a value or can reduce further. As infinitely looping programs can always reduce, they are deemed safe.

Safety is often shown using a type system, which consists of types and typing rules, capturing the set of values that the program terms can have. The types and typing rules are most commonly defined as inductive definitions and relations, respectively. For demonstration purposes, we consider a simple type system that has ground types (A_{g}) for integers and booleans, as well as a term type (A) for function types:

$$A_{\mathsf{g}} ::= \mathbf{Z} \mid \mathbf{B} \mid \ldots \qquad A ::= A_{\mathsf{g}} \mid A_1 \to A_2 \mid \ldots$$

The type system captures that an expression e has the type A with the typing judgement $\Gamma \vdash e : A$, where Γ is a typing context capturing the types of free variables. A selection of the typing rules for the type system is as follows:

$$\frac{}{\Gamma \vdash i : \mathbf{Z}} \qquad \frac{}{\Gamma \vdash b : \mathbf{B}} \qquad \frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x.\, e : A \to B}$$

$$\frac{\Gamma \vdash e_1 : A \to B \qquad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1\ e_2 : B} \qquad \frac{\Gamma \vdash e_1 : \mathbf{B} \qquad \Gamma \vdash e_2 : A \qquad \Gamma \vdash e_3 : A}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : A}$$

Safety of a closed expression e can then be established by checking that it is well-typed in the empty typing context ($\vdash e : A$), according to the typing rules.

Indeed, safety follows from type checking under the condition that the type system enjoys the property of *type safety*, stating that "well-typed programs cannot go wrong" [Milner 1978], which is formally captured as:

$$\vdash e : A \quad \text{implies} \quad \mathsf{safe}\ e \tag{2.1}$$

This property states that if a program is well-typed ($\vdash e : A$) it is also safe ($\mathsf{safe}\ e$). Proving that a type system satisfies the type safety property is most commonly done using using the technique of *progress* and *preservation* [Wright and Felleisen 1994].

- *progress*: well-typed programs are either values or they can take a step
- *preservation*: well-typed programs are still well-typed after taking a step

In particular, the properties are individually proven using induction, after which type safety follows directly.

Safety via Type Checking The following are examples of the type checking of two programs; The first being safe:

$$\dfrac{\dfrac{\overline{b : \mathbf{B} \vdash b : \mathbf{B}}\ \checkmark \quad \overline{b : \mathbf{B} \vdash 42 : \mathbf{Z}}\ \checkmark \quad \overline{b : \mathbf{B} \vdash 0 : \mathbf{Z}}\ \checkmark}{b : \mathbf{B} \vdash \mathtt{if}\ b\ \mathtt{then}\ 42\ \mathtt{else}\ 0 : \mathbf{Z}}}{\vdash \lambda b.\ \mathtt{if}\ b\ \mathtt{then}\ 42\ \mathtt{else}\ 0 : \mathbf{B} \to \mathbf{Z}}$$

And the second being unsafe:

$$\dfrac{\dfrac{\overline{b : \mathbf{B} \vdash b : \mathbf{B}}\ \checkmark \quad \overline{b : \mathbf{B} \vdash 42 : \mathbf{Z}}\ \checkmark \quad \overline{b : \mathbf{B} \vdash ()() : \mathbf{Z}}\ \times}{b : \mathbf{B} \vdash \mathtt{if}\ b\ \mathtt{then}\ 42\ \mathtt{else}\ ()() : \mathbf{Z}}}{\vdash \lambda b.\ \mathtt{if}\ b\ \mathtt{then}\ 42\ \mathtt{else}\ ()() : \mathbf{B} \to \mathbf{Z}}$$

In both programs we first abstract over the argument b of the boolean type $\mathbf{B}$, and then type check the if-else-expression by checking that the conditional argument b is of type $\mathbf{B}$, and that both branches have the same type (here the integer type $\mathbf{Z}$). The check on the conditional argument $b : \mathbf{B}$ passes in both examples, as per the typing context ($b : \mathbf{B}$). The check on the branches pass for the first example, as both branches are literal integers of type $\mathbf{Z}$. However, the second program is ill-typed, as the second branch $()()$ does not type check, since $()$ is not a function.

Limitations of Type Checking In both of the above examples the safety of the program corresponds directly to whether it is well-typed, but such a correspondence is not always the case, as stated in Section 1.4.2. Consider the following program which is safe, even though it is not well-typed:

$$\dfrac{\overline{\vdash \mathtt{true} : \mathbf{B}}\ \checkmark \quad \overline{\vdash 42 : \mathbf{Z}}\ \checkmark \quad \overline{\vdash ()() : \mathbf{Z}}\ \times}{\vdash \mathtt{if}\ \mathtt{true}\ \mathtt{then}\ 42\ \mathtt{else}\ ()() : \mathbf{Z}}$$

As before, the second branch of the program ()() is unsafe. However, the program as a whole is safe as the unsafe branch is never executed, since the conditional argument `true` holds vacuously. While conventional type systems are agnostic to such runtime behaviour, they can be captured when proving functional correctness, which in turn can prove that the program is in fact safe.

2.2.2 Functional Correctness

Functional correctness of a program is a property that is dependent on a given specification, which denotes the expected output of the program. Such a specification can be thought of as a predicate $\varphi \in \mathsf{Val} \to \mathsf{Prop}$. We call this predicate the postcondition, by which the correctness property can be captured formally as follows:[4]

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

The definition combines safety with postcondition validity, stating that the postcondition φ holds for any value v that the expression e reduces to. The correctness property subsumes our safety property, meaning that we have the following:

$$\mathsf{correct}\ (e, \varphi) \quad \text{implies} \quad \mathsf{safe}\ e \tag{2.2}$$

A formal approach to showing functional correctness is using a program logic, *e.g.,* Hoare logic [Hoare 1969], where *Hoare triples* $\{P\}\ e\ \{\Phi\}$ are used to ascribe a program e with a logical precondition $P \in \mathsf{iProp}$ and postcondition $\Phi \in \mathsf{Val} \to \mathsf{iProp}$. We often write $\{P\}\ e\ \{v.\, Q\}$ instead of $\{P\}\ e\ \{\lambda v.\, Q\}$, and $\{P\}\ e\ \{Q\}$, when $v = ()$. The propositions often range over a program logic iProp, that extends a meta-logic Prop, with connectives to describe the state of the program, *e.g.,* the locations of the heap. We often implicitly coerce predicates in the meta-logic φ into predicates of the program logic, *e.g.,* in $\{\mathsf{True}\}\ e\ \{\varphi\}$. A common variant of a program logic is separation logic, which is covered in Section 2.3.

Proving a Hoare triple is achieved by applying the rules of the program logic. As an example, the following are the rules for the conditional expression:

$$\textsc{Ht-if-true}\ \frac{\{P\}\ e_1\ \{\Phi\}}{\{P\}\ \texttt{if true then}\ e_1\ \texttt{else}\ e_2\ \{\Phi\}} \qquad \textsc{Ht-if-false}\ \frac{\{P\}\ e_2\ \{\Phi\}}{\{P\}\ \texttt{if false then}\ e_1\ \texttt{else}\ e_2\ \{\Phi\}}$$

Specifically, they capture that we are only concerned with the branch of the program that corresponds to the truth value of the boolean condition.

Connecting a Hoare-style program logic to the functional correctness of a program relies on a proof of *adequacy*, capturing that Hoare triples with meta-logic postconditions φ imply correctness up to that postcondition:

$$\{\mathsf{True}\}\ e\ \{\varphi\} \quad \text{implies} \quad \mathsf{correct}\ (e, \varphi) \tag{2.3}$$

[4] Note that we are not concerned with whether a program *terminates* (namely *total correctness*), but use the term correctness in place of the more apt term *partial correctness*, for brevity's sake.

The postcondition φ is restricted to the meta-logic fragment of the program logic as functional correctness $\mathsf{correct}\ (e, \varphi)$ is a meta-logic property. This may seem restrictive at first, but adequacy is only relevant for the specifications of the top-level expression of a program, *e.g.,* the main-function. As such, we are not concerned with properties about *e.g.,* the state of the heap, as it is soon to be deallocated.

The adequacy theorem is often obtained by modelling the Hoare triple in terms of the operational semantics of the language under consideration, such that it captures safety directly.

Functional Correctness via Hoare Logic An example of a (simplified) proof of functional correctness is the following derivation, which captures that the safe yet ill-typed program from the prior section is safe and that it results in an integer:

$$\frac{\overline{\{\mathsf{True}\}\ 42\ \{v.\ v \in \mathbb{Z}\}}\ \checkmark}{\{\mathsf{True}\}\ \texttt{if true then}\ 42\ \texttt{else}\ ()()\ \{v.\ v \in \mathbb{Z}\}}\ \textsc{Ht-if-true}$$

The derivation follows from Ht-if-true, by which we only need to focus on the first branch, as the conditional argument is vacuously `true`. After this step the program is a value, specifically the literal integer 42, for which the postcondition holds trivially. We can thus conclude that the program is safe by Equation (2.3) and Equation (2.2).

While more expressive than type checking, verifying whether there exists a valid derivation for a Hoare-style logic specification is often undecidable. Showing functional correctness via this technique thus often requires interactive effort, even when we only seek to prove safety, and not full functional correctness.

2.2.3 Semantic Typing

The techniques of type checking and program logic proofs may seem orthogonal, but they can be related through the technique of semantic typing [Milner 1978; Ahmed 2004; Ahmed et al. 2010] (contrasting syntactic typing, discussed thus far).

In a semantic type system the types and the typing judgement are given a semantic interpretation, respectively capturing what it means for a value to belong to a type, and that the program is safe to execute. The typing rules of a semantic type system are consequentially proven as lemmas, instead of being defined inductively.

The type safety of a semantic type system is a direct consequence of the definition of the typing judgement, as opposed to being proven using a technique such as progress and preservation by Wright and Felleisen [1994]. As a result, a semantic type system is *open*, where rules are abstractions capturing safety of specific term structures, in contrast to capturing a notion of being well-typed, which may then imply safety.

This openness effectively lets us add "ad-hoc" rules, whenever we reach a typing judgement with no corresponding typing rule, by proving it manually in the underlying model. This technique was applied by Jung et al. [2018a], who created a semantic type system for the Rust language, in which they manually prove safety of unsafe Rust libraries, *i.e.,* programs that deploy a complex form of sharing, such as

reference-counted pointers, which are beyond the scope of the type system. Achieving a similar result with the technique of progress and preservation is infeasible, as it would require typing rules for all mid-execution reductions.

Finally, as demonstrated by Appel et al. [2007] and Dreyer et al. [2010], building on top of an existing system, *e.g.,* Iris, a semantic type system can inherit the features of the existing system, *e.g.,* mutable references or recursion, its adequacy theorem, and any existing mechanisation effort.

A semantic type system corresponding to the simple syntactic type system shown in Section 2.2.1 can be obtained by building on top of the Hoare logic presented in Section 2.2.2. In particular, we define the types as a logical interpretation of the syntactic types $[\![\cdot]\!] \in \mathsf{Type} \to \mathsf{Val} \to \mathsf{iProp}$, where *e.g.,* the type of integers is defined as $[\![\mathbf{Z}]\!] \triangleq \lambda v.\, v \in \mathbb{Z}$. Drawing inspiration from the approach taken by Krebbers et al. [2017b], who built a semantic type system for a language similar to HeapLang, the typing judgement can be defined in terms of the Hoare triple:

$$\vDash e : A \triangleq \{\mathsf{True}\}\, e\, \{[\![A]\!]\} \tag{2.4}$$

The semantic typing rules are then proven as lemmas. As an example, the following is the typing rule for conditionals, which is expanded to the underlying Hoare triples:

$$\frac{\vDash e_1 : \mathbf{B} \qquad \vDash e_2 : A \qquad \vDash e_3 : A}{\vDash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : A} \quad \rightsquigarrow$$

$$\frac{\{\mathsf{True}\}\, e_1\, \{v.\, v \in \mathbb{B}\} \qquad \{\mathsf{True}\}\, e_2\, \{[\![A]\!]\} \qquad \{\mathsf{True}\}\, e_3\, \{[\![A]\!]\}}{\{\mathsf{True}\}\ \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3\, \{[\![A]\!]\}}$$

The proof of the underlying Hoare triple follows from case analysis on the boolean result of evaluating e_1, after which the rules Ht-if-true and Ht-if-false apply.

The connection to syntactic typing is captured by the fundamental lemma, often being a direct consequence of the semantic type system definitions, which states that the semantic type system subsumes the syntactic one:

$$\vdash e : A \quad \text{implies} \quad \vDash e : A \tag{2.5}$$

From the fundamental lemma, and the implications between Hoare triples, functional correctness, and safety, presented in Section 2.2.2, we then have the following subsumption relationship for the ground types A_{g}:[5]

$\vdash e : A_{\mathrm{g}}$	implies	(2.5)
$\vDash e : A_{\mathrm{g}}$	$\rightsquigarrow$	(2.4)
$\{\mathsf{True}\}\, e\, \{[\![A_{\mathrm{g}}]\!]\}$	implies	(2.3)
$\mathsf{correct}\ (e, [\![A_{\mathrm{g}}]\!])$	implies	(2.2)
$\mathsf{safe}\ e$		

With these definitions and implications, we can then prove safety of syntactically ill-typed programs via semantic typing.

[5] $\{\mathsf{True}\}\, e\, \{[\![A_{\mathrm{g}}]\!]\}$ implies $\mathsf{correct}\ (e, [\![A_{\mathrm{g}}]\!])$ for ground types A_{g}, as adequacy (2.3) only hold for meta-logic predicates, which the semantic interpretation of ground types commonly are.

Safety of Syntactically Ill-Typed Programs via Semantic Typing The semantic typing judgement for the safe yet ill-typed program presented in Section 2.2.1 correspond to the Hoare triple that we proved in Section 2.2.2:

$$\begin{array}{c} \vDash \texttt{if true then } 42 \texttt{ else } ()() : \mathbf{Z} \quad \rightsquigarrow \\ \{\mathsf{True}\}\ \texttt{if true then } 42 \texttt{ else } ()()\ \{v.\, v \in \mathbb{Z}\} \end{array}$$

The judgement can thus be used as a part of an otherwise syntactically checked typing derivation, even though it is internally ill-typed. This demonstrates how we can add ad-hoc rules for term structures that might otherwise not have a derivation in the syntactic system, *e.g.,* for the racy yet safe program presented in Section 1.4.2.

The ability to add such ad-hoc rules is how semantic typing allows for combining the simplicity of type checking with the expressivity of program logic proofs for showing safety. In particular, one can deploy type checking for the fragment of the program that does not rely on runtime behaviour, and manually prove the uncheckable remainder of the typing derivation in the underlying program logic.

The Foundational Approach In our semantic type system we take the *foundational approach* [Ahmed 2004], as similarly done by Jung et al. [2018a] for their semantic type system for Rust built on top of Iris. Taking the foundational approach means that we sidestep the syntactic type system, by not defining the types as inductive definitions. Instead we define the types as standalone predicates, *i.e.,* $\mathsf{Type} \in \mathsf{Val} \to \mathsf{iProp}$, rather than defining them as interpretations of syntactic types. Recovering proofs similar to the type checking of a syntactic type system can then be achieved by proving semantic typing rules (often called compatibility lemmas in the logical relations literature), corresponding to all of the rules that would otherwise be part of the syntactic type system.

By taking the foundational approach, we sidestep the effort of defining and mechanising a syntactic type system, and instead fully inherit the existing work of the underlying logic, which in our case is the features, adequacy, and mechanisation of the higher-order concurrent separation logic Iris.

2.3 The Iris Logic

In this section we cover the basics of the higher-order concurrent separation logic Iris [Jung et al. 2015, 2016] instantiated with HeapLang, and how it can be used to prove specifications of concurrent programs that safely share mutable data, such as the locked interleaving program presented in Section 2.1.3. An overview of grammar and inference rules of the logic can be found in Figure 2.6.

Program specifications in Iris can, alike standard Hoare and separation logics, be captured in terms of Hoare triples $\{P\}\, e\, \{\Phi\}$. Additionally, a proof of a Hoare triple in Iris similarly imply functional correctness for meta-logic postconditions $\varphi \in \mathsf{Val} \to \mathsf{Prop}$, as formally captured by Iris's adequacy theorem:

$$\{\mathsf{True}\}\, e\, \{\varphi\} \quad \text{implies} \quad \mathsf{correct}\ (e, \varphi)$$

The meta-logic of Iris is that of Coq, which is a higher-order propositional logic.

Grammar:

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

Ordinary affine separation logic:

$$\textsc{Affine}\quad P * Q \Rightarrow P$$

$$\textsc{Adjoint}\quad \frac{P * Q \vdash R}{P \vdash Q \mathrel{-\!\!*} R}$$

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

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

$$\textsc{Ht-csq}\quad \frac{\Box(P \mathrel{-\!\!*} P') \quad \{P'\}\, e\, \{w.\, Q'\} \quad \Box(\forall w.\, Q' \mathrel{-\!\!*} Q)}{\{P\}\, e\, \{w.\, Q\}}$$

$$\textsc{Ht-fork}\quad \frac{\{P\}\, e\, \{\mathsf{True}\}}{\{P\}\, \texttt{fork}\, \{e\}\, \{\mathsf{True}\}}$$

$$\textsc{Ht-bind}\quad \frac{\{P\}\, e\, \{v.\, Q\} \qquad \forall v.\, \{Q\}\, K[\,v\,]\, \{w.\, R\}}{\{P\}\, K[\,e\,]\, \{w.\, R\}}$$

Heap manipulation:

$$\textsc{Ht-alloc}\quad \{\mathsf{True}\}\, \texttt{ref}\, v\, \{\ell.\, \ell \mapsto v\}$$

$$\textsc{Ht-load}\quad \{\ell \mapsto v\}\, !\ell\, \{w.\, (w = v) * \ell \mapsto v\}$$

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

Recursion:

$$\triangleright\textsc{-intro}\quad P \Rightarrow \triangleright P$$

$$\textsc{Löb}\quad (\triangleright P \Rightarrow P) \Rightarrow P$$

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

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

$$\textsc{Ht-pers}\quad \frac{Q \Rightarrow \{P\}\, e\, \{w.\, R\}}{\{P * Q\}\, e\, \{w.\, R\}}\ \mathsf{persistent}(\mathrm{Q})$$

Figure 2.6: The grammar and a selection of rules of Iris for HeapLang.

The derivation of the ill-typed yet safe program presented in Section 2.2.2 is valid in Iris, as it follows from the rules Ht-if-true and Ht-val (up to some simplification):

$$\dfrac{\dfrac{}{\{\mathsf{True}\}\ 42\ \{v.\ v \in \mathbb{Z}\}}\ \textsc{Ht-val}}{\{\mathsf{True}\}\ \texttt{if true then}\ 42\ \texttt{else}\ ()()\ \{v.\ v \in \mathbb{Z}\}}\ \textsc{Ht-if-true}$$

The expressive power of Iris comes from the logical propositions $P, Q \in \mathsf{iProp}$ used in the Hoare triples. In this section we cover the features that are used in the remaining chapters of the thesis.

We first cover the separation logic properties of Iris (Section 2.3.1), which inherently guarantee that programs do not interfere with each other, as a result of Iris treating propositions affinely. This treatment is fundamental to how we reason about our programs, and is the foundation of how we model affine types in our semantic type system in Chapter 4.

We subsequently present the notion of persistent resources (Section 2.3.2), which captures how some propositions may be duplicated, and can thereby be distributed to multiple sub-proofs. This allows us to share *e.g.,* specifications of higher-order functions multiple times, and is what lets us model copyable types in our semantic type system in Section 4.3.2.

We then detail how Iris is a step-indexed separation logic (Section 2.3.3), which allows us to reason about recursive programs, and construct recursive predicates. This is essential to how we achieve the recursion of our session-type based protocols in Section 3.6.1, and consequentially how we obtain recursive session types of our semantic type system in Section 4.3.3.

We additionally demonstrate how Iris allows us to prove specifications of concurrent program, by using locks (Section 2.3.4) and ghost state (Section 2.3.5). These are both crucial parts of how we prove the high-level specifications of our channel implementation, which will be presented in Section 3.6.

We finally provide a more detailed overview of how we will use Iris to achieve the contributions of the remaining chapters (Section 2.3.6).

2.3.1 Separation Logic in Iris

Iris is an affine separation logic, in which propositions can be thought of as resources, that can be used at most once. The propositions for one range over the heap of the program. In particular, the points-to predicate $\ell \mapsto v$ asserts exclusive ownership of a location ℓ, while stating that it contains the value v. Meanwhile, the separating conjunction $P * Q$ captures that the propositions P and Q hold for disjoint parts of the heap. The exclusive ownership of a location ℓ can thereby only ever validly exist on one side of such a separating conjunction, and otherwise result in a contradiction:

$$(\ell \mapsto v) * (\ell \mapsto w) \Rightarrow \mathsf{False}$$

Finally, the separating implication $P \mathrel{-\!\!*} Q$ is similar to regular implication $P \Rightarrow Q$, for the separating conjunction, as captured by the Adjoint rule in Figure 2.6.

Separation logic allows for modular verification of subcomponents of a program, as specifications inherently guarantee that programs do not interfere with each other,

since their heap footprints are disjoint. We call this the separation property. The property is formally captured by the Ht-frame rule, which states that a frame of resources described by the proposition R, that is disjoint from the resources described by the proposition P, is preserved throughout execution of e, and thus also hold disjoint from the resources described by the proposition Q. As a result, we obtain small-footprint specifications, where we only describe the exact resources needed, as we can thread the disjoint frame of resources through our proofs with Ht-frame.

Examples of such small-footprint specifications are the rules Ht-alloc, Ht-load, and Ht-store. The Ht-alloc rule yields exclusive ownership of the newly allocated reference $\ell \mapsto v$. The rule Ht-load states that we can obtain the value stored in an exclusively owned reference. Finally, the rule Ht-store allows us to update the stored value of an exclusively owned reference. The exclusivity is crucial, as the view of the stored value could otherwise be inconsistent with other threads.

The separation property similarly ensures that concurrently running threads will not race for exclusive resources. This is made precise by the Ht-fork rule, which states that the subproof is modularly proven based on some disjoint part of the precondition. Finally, Iris is affine, meaning that resources can be dropped, as is standard for program logics over garbage collected languages. This property is made precise by the Affine rule.

Linked List Abstraction As an example of how separation logic can be used to implicitly capture disjointness of resources, consider the following definition, capturing what it means to be a linked list up to the interpretation predicate I:

$$\texttt{list\_interp}\ (I : T \to \mathsf{Val} \to \mathsf{iProp})\ (l : \mathsf{Loc})\ (\vec{x} : \mathsf{List}\ T) \triangleq \begin{cases} l \mapsto \texttt{None} & \text{if } \vec{x} = [\,] \\ \exists v_1\ l_2.\ I\ x_1\ v_1 * l \mapsto \texttt{Some}\ (v_1, l_2) * \texttt{list\_interp}\ I\ l_2\ \vec{x}_2 & \text{if } \vec{x} = [x_1] \cdot \vec{x}_2 \end{cases}$$

Here, T is a polymorphic type that is inferred from the I argument. We often use the notation $l \overset{\text{list}}{\mapsto}_I \vec{x}$ for $\texttt{list\_interp}\ I\ l\ \vec{x}$.

The definition is constructed as a fixpoint over a meta-level list $\vec{x}$. The definition captures the following:

1. The number of cells correspond to the length of the list $\vec{x}$
2. Separate exclusive ownership is asserted for each cell
3. The interpretation predicate I holds for each meta-value $x \in \vec{x}$, and its corresponding cell value v in the linked list

In particular, the predicate $l \overset{\text{list}}{\mapsto}_I \vec{x}$ asserts ownership of the linked list head-reference l, using the points-to connective. In the case that the list is empty ($\vec{x} = [\,]$), it asserts that the reference points to nothing $l \mapsto \texttt{None}$. If the list is non-empty ($\vec{x} = [x_1] \cdot \vec{x}_2$ for some values x_1 and $\vec{x}_2$), it asserts that the reference points to a cell $l \mapsto \texttt{Some}\ (v_1, l_2)$, and that the value v_1 can be interpreted as x_1 via $I\ x_1\ v_1$. Finally, the definition recursively captures separate ownership of the rest of the linked list $\texttt{list\_interp}\ I\ l_2\ \vec{x}_2$. The list predicate implicitly guarantees that the list does not form a cycle, as each of the value cells are guaranteed to be disjoint.

HT-LNIL
$\{\mathsf{True}\}\ \texttt{lnil}\ \{l.\, l \overset{\text{list}}{\mapsto}_I [\,]\}$

HT-LCONS
$\{l \overset{\text{list}}{\mapsto}_I \vec{x} * I\ x\ v\}\ \texttt{lcons}\ v\ l\ \{l \overset{\text{list}}{\mapsto}_I ([x] \cdot \vec{x})\}$

HT-LISNIL
$\{l \overset{\text{list}}{\mapsto}_I \vec{x}\}\ \texttt{lisnil}\ l\ \{b.\, b = (\vec{x} = [\,]) * l \overset{\text{list}}{\mapsto}_I \vec{x}\}$

HT-LSNOC
$\{l \overset{\text{list}}{\mapsto}_I \vec{x} * I\ x\ v\}\ \texttt{lsnoc}\ l\ v\ \{l \overset{\text{list}}{\mapsto}_I (\vec{x} \cdot [x])\}$

HT-LPOP
$\{l \overset{\text{list}}{\mapsto}_I ([x] \cdot \vec{x})\}\ \texttt{lpop}\ l\ \{w.\, I\ x\ w * l \overset{\text{list}}{\mapsto}_I \vec{x}\}$

Figure 2.7: Specifications of a selection of the functions for the mutable linked list implementation presented in Section 2.1.4.

The specifications of a selection of the the linked list functions is found in Figure 2.7. To demonstrate how we can prove such specifications of programs with multiple sub-expression, consider the specification for `lcons`, which is defined as follows:

$$\texttt{lcons}\ v\ l := l \leftarrow \texttt{Some}\ (v, \texttt{ref}\ (!\, l))$$

We first need to do case analysis on the meta-list $\vec{x}$, to obtain the reference ownership of l, which is present in both the empty and non-empty case. As the proof of both cases are similar, we only consider the empty case for simplicity sake:

$$\{l \mapsto \texttt{None} * I\ x\ v\}\ \texttt{lcons}\ v\ l\ \{l \overset{\text{list}}{\mapsto}_I [x]\}$$

As the program consists of multiple sub-expressions, *e.g.*, $!\, l$, we use the HT-BIND rule to string the sub-proofs of each sub-expression together. We thus start at the first sub-expression $!\, l$, being at the head-position of the program, captured by the surrounding evaluation context $K \triangleq (l \leftarrow \texttt{Some}\ (v, \texttt{ref}\ \bullet))$:

$$\{l \mapsto \texttt{None} * I\ x\ v\}\ (l \leftarrow \texttt{Some}\ (v, \texttt{ref}\ \bullet))[\,!\, l\,]\ \{l \overset{\text{list}}{\mapsto}_I [x]\}$$

We use the rule HT-LOAD to resolve the reference deallocation $!\, l$, which returns `None`. We then continue to the next sub-expression, being the reference allocation `ref None`:

$$\{l \mapsto \texttt{None} * I\ x\ v\}\ (l \leftarrow \texttt{Some}\ (v, \bullet))[\texttt{ref None}\,]\ \{l \overset{\text{list}}{\mapsto}_I [x]\}$$

Here we use the rule HT-ALLOC to resolve the reference allocation `ref None`, which returns a new location l', for which we obtain $l' \mapsto \texttt{None}$. We then continue to the final sub-expression $l \leftarrow \texttt{Some}\ (v, l')$:

$$\{l \mapsto \texttt{None} * l' \mapsto \texttt{None} * I\ x\ v\}\ \bullet\, [\, l \leftarrow \texttt{Some}\ (v, l')\,]\ \{l \overset{\text{list}}{\mapsto}_I [x]\}$$

We resolve the step with the HT-STORE rule, which updates the reference predicate of l to $l \mapsto \texttt{Some}\ (v, l')$. We finally have:

$$\{l \mapsto \texttt{Some}\ (v, l') * l' \mapsto \texttt{None} * I\ x\ v\}\ ()\ \{l \overset{\text{list}}{\mapsto}_I [x]\}$$

This final specification then follows directly from the definition of $l \overset{\text{list}}{\mapsto}_I \vec{x}$.

2.3.2 Persistent Resources in Iris

As Iris propositions are treated as affine resources they can by default be used at most once. However, some propositions, *e.g.,* equality, do not assert any properties about resource ownership, and are thus safe to reuse multiple times. It therefore makes sense to differentiate such propositions from affine propositions, as we might want to distribute them to multiple subproofs. In Iris we call these propositions persistent, which essentially mean that they can be duplicated. This is made precise by the $\Box$ modality, which enjoys the following rules:

$$\mathsf{persistent}(P) \triangleq P \Rightsquigarrow \Box P \qquad \underset{\Box P \Rightsquigarrow (\Box P) * (\Box P)}{\Box\text{-DUP}} \qquad \underset{\Box P \Rightsquigarrow P}{\Box\text{-ELIM}}$$

A proposition P is considered to be persistent ($\mathsf{persistent}(P)$), if it can be freely extended with the $\Box$ modality, as captured by its definition. The rule $\Box$-DUP captures that propositions under the $\Box$ modality can be duplicated freely. Finally, The rule $\Box$-ELIM captures that the $\Box$ modality can always be eliminated, letting us obtain the persistent proposition. A persistent proposition can then be freely extended with the always modality $\Box$, after which it can be duplicated with $\Box$-DUP and $\Box$-ELIM:

$$\begin{aligned} P &\Rightsquigarrow && (\mathsf{persistent}(P)) \\ \Box P &\Rightsquigarrow && (\Box\text{-DUP}) \\ \Box P * \Box P &\Rightsquigarrow && (\Box\text{-ELIM}) \\ P * P \end{aligned}$$

The persistent propositions presented thus far are $t = u$, True, False, and $\Box P$. Moreover, the persistent modality commutes with conjunctions ($P \wedge Q$, $P * Q$), disjunction ($P \vee Q$), universal and existential quantification ($\forall x.\, P$, $\exists x.\, P$). The set of persistent propositions is therefore closed under these connectives.

Weakest Precondition We have thus far presented the program logic of Iris in terms of Hoare triples. However, in Coq the program logic of Iris is actually defined in terms of the weakest precondition $\mathsf{wp}\ e\ \{\Phi\}$. The weakest precondition captures that the expression e does not get stuck, and that if it reduces to a value v, then $\Phi\ v$ holds. The actual adequacy statement of Iris is thus:

$$\mathsf{wp}\ e\ \{\varphi\} \quad \text{implies} \quad \mathsf{correct}\ (e, \varphi)$$

Like the Hoare triples, the weakest preconditions are first-class members of the logic, and can thus be nested.

The Hoare triples of Iris are used as an intuitive way of defining program specifications and rules, and is defined in terms of the weakest precondition:

$$\{P\}\ e\ \{\Phi\} \triangleq \Box(P \mathrel{-\!\!*} \mathsf{wp}\ e\ \{\Phi\})$$

In particular, they state that under the assumption that the precondition P holds, as captured by the separating implication $-\!\!*$, the weakest precondition for the expression e and postcondition Φ holds. The definition is wrapped in the $\Box$ modality to capture

that Hoare triples do not depend on external resources, beyond what is provided in their precondition P. This in turn makes them persistent. This is crucial as Hoare triples denote specifications of programs, which can be executed repeatedly, and should therefore be duplicable.

We omit the definition and rules for the weakest precondition. The rules are similar in spirit to the corresponding Hoare triple rules. For more details on the definition of the weakest precondition of Iris we refer the interested reader to Jung et al. [2018b, Section 7.3].

For presentation purposes, we primarily use the Hoare triples in Chapter 3, while we instead use weakest preconditions in Chapter 4.

2.3.3 Step-Indexing in Iris

Iris is a step-indexed separation logic [Appel et al. 2007; Hobor et al. 2008; Svendsen et al. 2010; Dreyer et al. 2010], where propositions are implicitly indexed by a natural number called a step-index. The step-index allows us to create structurally recursive propositions, in which the number gets smaller. Finally, the step-index is associated with the number of steps taken by a program, to capture that propositions only hold after a certain number of steps. As a result, we can differentiate between propositions that hold now, and those that hold later. The step-indexing of Iris gives us:

1. A way of constructing recursive predicates, with the guarded fixpoint operator $\mu x.\, t$, which *e.g.,* lets us define infinite channel protocols as demonstrated in Section 3.6.2
2. Löb induction, which lets us reason about recursive functions and predicates, demonstrated momentarily
3. Impredicative invariants, which lets us prove high-level specifications of concurrent programs that safely share exclusive resources, in terms of custom abstract predicates, which are seemingly first-class citizens of the logic.

The step-indexing of Iris manifests with the later modality $\triangleright$, which is used to guard proposition *e.g.,* $\triangleright P$, capturing that they hold after one step of computation. This property is demonstrated in the Ht-rec rule:

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

The rule states that the guarded precondition $\triangleright P$ is stripped of its later, when the function application has been resolved, *i.e.,* when a step has been taken.

Löb Induction The step-indexing of Iris allows for reasoning about recursive programs via Löb induction, an induction principle over the steps of the program. The induction principle manifests with the Löb rule:

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

The rule states that when proving a proposition P, we can assume that the proposition holds one program step later $\triangleright P$.

To demonstrate Löb induction, consider the following infinitely looping program:

$$\texttt{loop\_prog}\ () := \texttt{loop\_prog}\ ()$$

The program repeatedly calls itself recursively, taking a single step (the application of the function), at each iteration. We want to show that the program is safe, even though it never terminates. This can be captured with a Hoare triple where the postcondition is False. The proof of the program is as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\{\mathsf{True}\}\ \texttt{loop\_prog}\ ()\ \{\mathsf{False}\} \Rrightarrow \{\mathsf{True}\}\ \texttt{loop\_prog}\ ()\ \{\mathsf{False}\}}\ \checkmark}{\{(\{\mathsf{True}\}\ \texttt{loop\_prog}\ ()\ \{\mathsf{False}\})\}\ \texttt{loop\_prog}\ ()\ \{\mathsf{False}\}}\ \textsc{Ht-pers}}{\{\triangleright(\{\mathsf{True}\}\ \texttt{loop\_prog}\ ()\ \{\mathsf{False}\})\}\ \texttt{loop\_prog}\ ()\ \{\mathsf{False}\}}\ \textsc{Ht-rec}}{\{\mathsf{True}\}\ \texttt{loop\_prog}\ ()\ \{\mathsf{False}\}}\ \textsc{Löb, Ht-pers}$$

We first use the Löb rule, to obtain the Löb induction hypothesis, along with Ht-pers, to move the guarded specification induction hypothesis into the Hoare triple precondition. We then strip the later of the hypothesis by resolving the function application with Ht-rec. Finally, the proof follows immediately by moving the specification out of the precondition with Ht-pers.

2.3.4 Reasoning about Concurrent Programs in Iris

The separation property of separation logic guarantees that resources are disjoint. This lets us reason about concurrently running threads that work on disjoint parts of the heap, as they are guaranteed to not interfere with each other.

However, we often wish to reason about concurrent threads that safely access overlapping memory footprints. Consider the `locked_interleaving_example` program from Section 2.1.3, which safely share a location ℓ by protecting it using a lock around the critical sections of the parallel threads in which ℓ is accessed:

$$\begin{array}{l}\texttt{locked\_interleaving\_example}\ \ell := \\ \quad \texttt{let}\ lk := \texttt{new\_lock}\ ()\ \texttt{in} \\ \quad \left(\begin{array}{l}\texttt{acquire}\ lk; \\ \quad \texttt{let}\ x := !\,\ell\ \texttt{in} \\ \quad \ell \leftarrow x + 2; \\ \texttt{release}\ lk\end{array} \;\middle\|\; \begin{array}{l}\texttt{acquire}\ lk; \\ \quad \texttt{let}\ x := !\,\ell\ \texttt{in} \\ \quad \ell \leftarrow x + 2; \\ \texttt{release}\ lk\end{array}\right); \\ \quad \texttt{acquire}\ lk\end{array}$$

For starters we simply want to prove safety, specified as follows:

$$\{\ell \mapsto x\}\ \texttt{locked\_interleaving\_example}\ \ell\ \{\mathsf{True}\}$$

To prove such a specification, we can use the high-level specifications for the lock and parallel composition abstractions shown in Figure 2.8. These specifications are inspired by the lock implementation by Jourdan and Krebbers [2018, Exercise 3] and the implementation of the parallel composition by Iris Development Team [2021, HeapLang], who also prove their respective specifications on top of Iris.

Specifications of locks:

Ht-newlock
$\{R\}\ \texttt{new\_lock}\ ()\ \{lk.\ \mathsf{is\_lock}\ lk\ R\}$

Ht-acquire
$\{\mathsf{is\_lock}\ lk\ R\}\ \texttt{acquire}\ lk\ \{R\}$

Ht-release
$\{\mathsf{is\_lock}\ lk\ R * R\}\ \texttt{release}\ lk\ \{\mathsf{True}\}$

lock-pers
$\mathsf{persistent}(\mathsf{is\_lock}\ lk\ R)$

Specifications of parallel composition:

Ht-par

$$\frac{\{P_1\}\ e_1\ \{\Phi_1\} \qquad \{P_2\}\ e_2\ \{\Phi_2\}}{\{P_1 * P_2\}\ e_1\ ||\ e_2\ \{w.\ \exists w_1, w_2.\ w = (w_1, w_2) * \Phi_1\ w_1 * \Phi_2\ w_2\}}$$

Figure 2.8: Specifications of the parallel composition and lock implementations.

The rule Ht-par for parallel composition ($e_1\ ||\ e_2$) states that we can prove the threads e_1 and e_2 separately, by splitting the precondition into two disjoint sets of resources P_1 and P_2. As a result we obtain the postconditions Φ_1 and Φ_2 for the respective threads and their resulting values w_1 and w_2.

The specifications of the lock operate on the *abstract predicate* $\mathsf{is\_lock}\ lk\ R$, which conceptually captures that the value lk is a lock which governs the proposition R. Additionally, the predicate $\mathsf{is\_lock}\ lk\ R$ is persistent, allowing us to duplicate it and share it between multiple threads. Finally, the predicate $\mathsf{is\_lock}\ lk\ R$ is built on top of Iris's impredicative invariants, and can therefore be thought of as a first-class citizen of the logic, which lets us put locks inside of locks.

The rule Ht-newlock states that we can allocate a new lock by giving up R, in turn giving us the abstract predicate $\mathsf{is\_lock}\ lk\ R$ for the return value lk. We can then access the locked proposition R with Ht-acquire, which waits for the lock to be free, to then acquire it, after which we obtain R. Finally, we can release the lock with Ht-release, under the condition that we give back the locked proposition R.

The proof of the `locked_interleaving_example` safety specification is carried out using Ht-bind to string the sub-proofs of the individual sub-expressions together. The individual steps are carried out as follows:

- Allocate the lock lk, protecting the exclusive reference, to get the abstract predicate $P_{lk} \triangleq \mathsf{is\_lock}\ lk\ (\exists x.\ \ell \mapsto x)$, using Ht-newlock
- Duplicate P_{lk} using lock-pers, $\square$-dup, and $\square$-elim
- Distribute P_{lk} to the subproofs of the parallel threads with Ht-par
- For each of the parallel threads:
  - Acquire the lock to obtain the exclusive reference $\ell \mapsto x$ with Ht-acquire
  - Deallocate and update the reference with Ht-load and Ht-store
  - Release the lock by putting the updated exclusive reference $\ell \mapsto (x + 2)$ back into the lock (with x instantiated as $x + 2$) with Ht-release
- Acquire the lock, and throw away P_{lk} with Ht-acquire and Affine

In P_{lk} we existentially quantify the stored value, as it must remain unchanged outside of the critical sections. This limitation restricts us from proving a stronger functional correctness specification, namely that the program adds exactly 4 to the stored value, using only the lock and parallel composition specifications. However, we can achieve such a proof with Iris's notion of *ghost state*, which lets us keep track of the changes made to resources owned by invariants.

2.3.5 Ghost State in Iris

Ghost state is similar in spirit to heap state, as it carries ownership, but instead of describing the state of the heap it describes the state of proof-related variables, which are not a part of the actual program [Cohen et al. 2009; Krishnaswami et al. 2012].

Ghost state manifests as ghost resources, of the form $\boxed{x}^{\gamma}$, where γ is the resource identifier, and x is its value. The values governed by ghost resources can take many forms, each with an individual set of rules for constructing and updating the resources. We call these sets of rules ghost theories. Iris comes equipped with a library of so-called resource algebras, for constructing various kinds of ghost theories. For the purpose of this thesis we do not cover how we construct our ghost theories, and instead simply present their rules. We refer the interested reader to Jung et al. [2018b, Section 3.1].

To prove functional correctness of the `locked_interleaving_example` program we consider a ghost theory consisting of two resources $\boxed{\bullet\, x}^{\gamma}$ and $\boxed{\circ_{\pi}\, x}^{\gamma}$. The resource $\boxed{\bullet\, x}^{\gamma}$, which we call the authoritative piece, conceptually captures the global view of $x \in \mathbb{Z}$. The resource $\boxed{\circ_{\pi}\, y}^{\gamma}$, which we call the permission piece, conceptually captures the permission to update the value recorded by the authoritative piece. Here, $y \in \mathbb{Z}$ is a local view of the updates that has been made to the authoritative piece in the presence of this permission piece. The fraction $\pi \in (0, 1]$ denotes how big a part of the full view that the local view represents. These properties are made precise by the following rules:

$$\textsc{Vs-csq}\quad \frac{P \Rrightarrow_{\mathcal{E}} P' \qquad \{P'\}\, e\, \{w.\, Q'\} \qquad (\forall w.\, Q' \Rrightarrow_{\mathcal{E}} Q)}{\{P\}\, e\, \{w.\, Q\}}$$

$$\textsc{Vs-fracAuth-alloc}\quad \mathsf{True} \Rrightarrow_{\mathcal{E}} \exists \gamma.\, \boxed{\bullet\, x}^{\gamma} * \boxed{\circ_1\, x}^{\gamma}$$

$$\textsc{Vs-fracAuth-add}\quad \boxed{\bullet\, x}^{\gamma} * \boxed{\circ_{\pi}\, y}^{\gamma} \Rrightarrow_{\mathcal{E}} \boxed{\bullet\, (x+z)}^{\gamma} * \boxed{\circ_{\pi}\, (y+z)}^{\gamma}$$

$$\textsc{fracAuth-op}\quad \boxed{\circ_{\pi_1+\pi_2}\, (x+y)}^{\gamma} \mathrel{**} \boxed{\circ_{\pi_1}\, x}^{\gamma} * \boxed{\circ_{\pi_2}\, y}^{\gamma} \qquad \textsc{fracAuth-eq}\quad \boxed{\bullet\, x}^{\gamma} * \boxed{\circ_1\, y}^{\gamma} \mathrel{-\!\!*} x = y$$

Here the rule Vs-csq states that we can always update the ghost state before or after the execution of any program. That is by employing the viewshift $\Rrightarrow_{\mathcal{E}}$, which for the purpose of this thesis can be thought of as an implication for the ghost state. We can thus always allocate an authoritative piece $\boxed{\bullet\, x}^{\gamma}$ and a full-fraction permission piece $\boxed{\circ_1\, x}^{\gamma}$, with some freely picked value x, and a fresh identifier γ, by using Vs-fracAuth-alloc. The rule Vs-fracAuth-add captures that we can update the

authoritative piece $\boxed{\bullet\ x}^{\gamma}$ in the presence of a permission piece $\boxed{\circ_{\pi}\ y}^{\gamma}$ with any partial fraction π. The rule FRACAUTH-OP lets us split and rejoin the permission pieces into and from smaller fractions, by similarly splitting the local view that they have recorded. This lets us distribute permission pieces to multiple threads, and later recover the complete record of the changes that they have recorded. Finally, the FRACAUTH-EQ rule lets us assert that the value stored in the authoritative piece corresponds to the value recorded by a permission piece with the full fraction.

With this ghost theory in hand we can prove the following functional specification of the `locked_interleaving_example` program:

$$\{\ell \mapsto x\}\ \texttt{locked\_interleaving\_example}\ \ell\ \{\ell \mapsto (x+4)\}$$

We again string together the sub-proofs of the sub-expressions using HT-BIND, where the individual sub-proofs are proven as follows:

- Allocate the ghost variables $\boxed{\bullet\ 0}^{\gamma}$ and $\boxed{\circ_1\ 0}^{\gamma}$ which govern the amount added to the stored value x (initially being 0), using VS-CSQ and VS-FRACAUTH-ALLOC
- Allocate the lock lk, protecting the exclusive reference $\ell \mapsto x$ along with the authoritative piece $\boxed{\bullet\ 0}^{\gamma}$, to get the abstract predicate for the lock $P_{lk} \triangleq \mathsf{is\_lock}\ lk\ (\exists y.\, \ell \mapsto (x+y) * \boxed{\bullet\ y}^{\gamma})$, using HT-NEWLOCK
- Duplicate P_{lk} using LOCK-PERS, $\Box$-DUP, and $\Box$-ELIM
- Split the permission piece $\boxed{\circ_1\ 0}^{\gamma}$ into $\boxed{\circ_{\frac{1}{2}}\ 0}^{\gamma} * \boxed{\circ_{\frac{1}{2}}\ 0}^{\gamma}$, using FRACAUTH-OP
- Distribute P_{lk} and one of the permission pieces $\boxed{\circ_{\frac{1}{2}}\ 0}^{\gamma}$ to each subproof of the parallel threads with HT-PAR
- For each of the parallel threads:
  - Acquire the exclusive reference $\ell \mapsto (x+y)$ and authoritative piece $\boxed{\bullet\ y}^{\gamma}$ with HT-ACQUIRE
  - Deallocate and update the reference with HT-LOAD and HT-STORE
  - Update the ghost variables to record the value change, from $\boxed{\bullet\ y}^{\gamma} * \boxed{\circ_{\frac{1}{2}}\ 0}^{\gamma}$ to $\boxed{\bullet\ (y+2)}^{\gamma} * \boxed{\circ_{\frac{1}{2}}\ 2}^{\gamma}$, with VS-FRACAUTH-ADD
  - Release the lock by putting the updated exclusive reference $\ell \mapsto (x+y+2)$ and the authoritative piece $\boxed{\bullet\ (y+2)}^{\gamma}$ (with y instantiated as $y+2$) back into the lock with HT-RELEASE
- Rejoin the permission pieces returned by HT-PAR $\boxed{\circ_{\frac{1}{2}}\ 2}^{\gamma} * \boxed{\circ_{\frac{1}{2}}\ 2}^{\gamma}$ to $\boxed{\circ_1\ 4}^{\gamma}$
- Acquire the exclusive reference $\ell \mapsto (x+y)$ and authoritative piece $\boxed{\bullet\ y}^{\gamma}$ with HT-ACQUIRE
- Infer that the value stored in ℓ is $x+4$ from $\boxed{\bullet\ y}^{\gamma} * \boxed{\circ_1\ 4}^{\gamma}$, using FRACAUTH-EQ

We treat the final `acquire` lk as a means of deallocating the lock, to re-obtain the protected resources. We could circumvent the need for this step with another set of specifications for the lock, that keep track of how many times P_{lk} has been duplicated [Hobor et al. 2008; Bizjak et al. 2019]. This would allow us to re-obtain the protected resources by forgetting the lock, in the presence of all of the duplicated

fragments. However, keeping track of the lock fragments adds additional complexity to our proofs, and would not allow us to model the mutex types in our semantic type system Section 4.3.6.

2.3.6 Concluding Remarks

We have shown how Iris can be used to prove high-level specifications of implemented abstractions, such as locks (Section 2.3.4). These specifications are based on abstract predicates, *e.g.,* is_lock $lk\ R$, constructed such that we can think of them as first-class primitives, akin to $\ell \mapsto v$. As a result, the abstractions seamlessly integrate with each other, and can be nested.

We take a similar approach in Chapter 3, to obtain high-level specifications for the binary channels for message passing presented in Section 2.1.5. The specifications of the channels are based on an abstract predicate that captures exclusive ownership of a channel endpoint, along with the channels compliance with one of our novel dependent separation protocols. The abstract predicate is constructed such that it can be thought of as a first-class citizen of the logic. As a result, we can nest protocols, allowing us to describe the exchange of channels over channels, which we demonstrate in Section 3.2. Additionally, it makes the channels seamlessly integrate with locks, as we can put channels in locks, and vice versa, which we demonstrate in Section 3.4.

We have additionally given an account of various Iris features, such as basic separation logic (Section 2.3.1), persistency (Section 2.3.2), and step-indexing (Section 2.3.3). These Iris features give rise to many of the features of the semantic type system, which we present in Chapter 4. For example, the affine properties of the type system are a direct result of the affine separation logic of Iris, while the copyable types are obtained based on the persistency of the type predicates.

Finally, we have shown how we can use Iris's ghost state mechanism to prove properties about programs that share exclusive resources via locks (Section 2.3.5). We employ a similar approach to prove the specifications of various programs that combine message passing with locks. In particular, we prove the functional correctness of a load-balancing mapper in Section 3.4. We prove the functional correctness of a map-reduce algorithm in Section 3.5. Finally, we do a manual typing proof of a producer-consumer-based computation client that operates on a single channel endpoint in parallel in Section 4.4.

Chapter 3

Actris: Session-Type Based Reasoning in Separation Logic

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

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

3.1 Introduction

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

with close to constant up-time, using lightweight processes that can be spawned by the hundreds of thousands and that communicate via asynchronous message passing.

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

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

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

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

Program specifications in Actris are written in an impredicative higher-order concurrent separation logic built on top of the Iris framework [Jung et al. 2015; Krebbers et al. 2017a; Jung et al. 2016, 2018b]. In addition to the usual features of Iris, Actris provides a notion of *dependent separation protocols* to reason about message passing over channels, inspired by the affine variant [Mostrous and Vasconcelos 2014] of binary session types [Honda et al. 1998]. We show that dependent separation protocols integrate seamlessly with other concurrency paradigms, allow delegation of resources, support channel sharing over multiple concurrent threads using locks, and more.

3.1.1 Message Passing in Concurrent Separation Logic

Over the last decade, there has been much work on extensions of concurrent separation logic with reasoning principles for message passing [Oortwijn et al. 2016; Francalanza et al. 2011; Lozes and Villard 2012; Craciun et al. 2015]. These logics typically include some form of mechanism for writing protocol specifications in a high-level manner. Unfortunately, these logics have shortcomings in terms of expressivity. Most importantly, they cannot be used to reason about programs that combine message passing with other programming and concurrency paradigms, such as higher-order functions, fine-grained shared-memory concurrency, and locks.

In a different line of work, researchers have developed expressive extensions of concurrent separation logic that do support proving strong specifications of programs involving some or all combinations of the aforementioned programming and concurrency paradigms. Examples of such logics are TaDA [da Rocha Pinto et al. 2014], iCAP [Svendsen and Birkedal 2014], Iris [Jung et al. 2015], FCSL [Nanevski et al.

2014], and VST [Appel 2014]. However, only a few variants and extensions of these logics address message-passing concurrency.

First, there has been work on the use of separation logic to reason about programs that communicate via message passing over a network. The reasoning principles in such logics are geared towards different programming patterns than the ones used in high-level languages like Erlang, Elixir, Go, and Scala. Namely, on networks all data must be serialised, and packets can be lost or delivered out of order. In high-level languages messages cannot get lost, are ensured to be delivered in order, and are allowed to contain many types of data, including functions, references, and even channel endpoints. Two examples of network logics are Disel by Sergey et al. [2018] and Aneris by Krogh-Jespersen et al. [2020]. Second, there has been work on the use of separation logic to prove compiler correctness of high-level message-passing languages. Tassarotti et al. [2017] verified a small compiler of a session-typed language into a language where channel buffers are modelled on the heap.

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

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

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

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

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

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

3.1.2 Actris 1.0: Dependent Separation Protocols

Instead of using STSs, Actris extends separation logic with a new notion called *dependent separation protocols*. This notion is inspired by the session type community,

pioneered by Honda et al. [1998], where channel endpoints are given types that describe the expected exchanges. Using binary session types, the channels c and c' in the program $prog_1$ in Section 3.1.1 would have the types $c :$ **?**$\mathbb{Z}$.**end** and $c' :$ **!**$\mathbb{Z}$.**end**, where **!**T and **?**T denotes that a value of type T is sent or received, respectively. Moreover, the types of the channels c and c' are *duals*—when one does a send the other does a receive, and *vice versa*.

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

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

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

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

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

In this program, the main thread sends the number 40 to the forked-off thread, which then adds 2 to it, and sends it back. This program has the same specification as $prog_1$, while we change the dependent separation protocol as follows (we omit the dependent separation protocol for the dual endpoint c'):

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

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

$2\rangle\{\mathsf{True}\}.\,\mathbf{end}$ where x is free, which means the value of x must be known when the protocol is created.

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

$$\begin{aligned} \mathit{prog}_3 := {} & \mathtt{let}\ (c, c') := \mathtt{new\_chan}\ ()\ \mathtt{in} \\ & \mathtt{fork}\ \{\mathtt{let}\ \ell := \mathtt{recv}\ c'\ \mathtt{in}\ \ell \leftarrow\ !\,\ell + 2;\ \mathtt{send}\ c'\ ()\}\,; \\ & \mathtt{let}\ \ell := \mathtt{ref}\ 40\ \mathtt{in}\ \mathtt{send}\ c\ \ell;\ \mathtt{recv}\ c;\ !\,\ell \end{aligned}$$

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

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

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

The type system by Bocchi et al. [2010] cannot verify this program because it does not support mutable state, while Actris can verify the program because it is a separation logic. The logic by Craciun et al. [2015] cannot verify this program because it does not support dependent protocols, which are crucial here as they make it possible to delay picking the location ℓ used in the protocol until the send operation is performed.

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

$$\begin{aligned} \mathit{prog}_4 := {} & \mathtt{let}\ (c, c') := \mathtt{new\_chan}\ ()\ \mathtt{in} \\ & \mathtt{fork}\ \big\{\mathtt{let}\ \mathit{go}\ () := \big(\mathtt{send}\ c'\ (\mathtt{recv}\ c' + 2);\ \mathit{go}\ ()\big)\ \mathtt{in}\ \mathit{go}\ ()\big\}\,; \\ & \mathtt{send}\ c\ 18;\ \mathtt{let}\ x := \mathtt{recv}\ c\ \mathtt{in} \\ & \mathtt{send}\ c\ 20;\ \mathtt{let}\ y := \mathtt{recv}\ c\ \mathtt{in}\ x + y \end{aligned}$$

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

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

This protocol expresses that it is possible to make repeated exchanges with the forked-off thread to increment a number by 2. The fact that the variable x is bound in the protocol is once again crucial—it allows the use of different numbers for each exchange.

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

$$\begin{aligned} \mathit{prog}_5 := {} & \texttt{let}\ (c_1, c_1')\ :=\ \texttt{new\_chan}\ ()\ \texttt{in} \\ & \texttt{fork}\ \{\texttt{let}\ c := \texttt{recv}\ c_1'\ \texttt{in}\ \texttt{let}\ y := \texttt{recv}\ c_1'\ \texttt{in}\ \texttt{send}\ c\ y;\ \texttt{send}\ c_1'\ ()\}; \\ & \texttt{let}\ (c_2, c_2')\ :=\ \texttt{new\_chan}\ ()\ \texttt{in} \\ & \texttt{fork}\ \{\texttt{let}\ x := \texttt{recv}\ c_2'\ \texttt{in}\ \texttt{send}\ c_2'\ (x+2)\}; \\ & \texttt{send}\ c_1\ c_2;\ \texttt{send}\ c_1\ 40;\ \texttt{recv}\ c_1;\ \texttt{recv}\ c_2 \end{aligned}$$

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

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

The first protocol states that the initial value sent must be a channel endpoint with the protocol used in prog_1, meaning that the main thread must give up the ownership of the channel endpoint c_2, thereby delegating it. It then expects a value y to be sent, and finally to receive a notification $()$, along with ownership of the channel c_2, which has since taken one step by sending y.

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

$$\begin{aligned} \mathit{prog}_6 := {} & \texttt{let}\ (c, c')\ :=\ \texttt{new\_chan}\ ()\ \texttt{in} \\ & \texttt{fork}\ \{\texttt{let}\ f := \texttt{recv}\ c'\ \texttt{in}\ \texttt{send}\ c'\ (\lambda().\, f() + 2)\}; \\ & \texttt{let}\ \ell := \texttt{ref}\ 40\ \texttt{in}\ \texttt{send}\ c\ (\lambda().\, !\,\ell);\ \texttt{recv}\ c\ () \end{aligned}$$

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

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

The send constructor ($\mathbf{!}$) 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. To send the function, the main thread would let $P \triangleq \ell \mapsto 40$ and $Q(v) \triangleq (v = 40)$, and prove $\{P\}\,(\lambda().\, !\,\ell)\ ()\ \{Q\}$. This example demonstrates that the state space of dependent separation protocols can be higher-order—it is indexed by the precondition P and postcondition Q of f—which means that they do not have to be agreed upon when creating the protocol, masking the internals of the function from the forked-off thread.

It is worth noting that using dependent recursive protocols it is possible to keep track of a history of what actions have been performed, which, as is shown in Section 3.4, is especially useful when combining channels with locks.

3.1.3 Actris 2.0: Subprotocols

While Actris 1.0's notion of dependent separation protocols is expressive enough to capture advanced exchanges, as indicated by the examples in the previous section, they are more restrictive than necessary due to their dual nature. The dual nature of dependent separation protocols requires that:

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

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

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

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

Support for type checking such programs has been studied in the session type community, namely in the context of *asynchronous session subtyping* [Mostrous et al. 2009; Mostrous and Yoshida 2015], in which a subtyping relation $S_1 <: S_2$ is defined, capturing that the session type S_2 can be used in place of S_1 when type checking a program. The relation captures that sends can be swapped ahead of receives $\mathbf{?}T.\mathbf{!}U.S <: \mathbf{!}U.\mathbf{?}T.S$. We refer to this as messages being sent *ahead of* the receives.

In this paper, we show that dependent separation protocols are compatible with the idea of asynchronous session subtyping. This gives rise to **Actris 2.0** that supports so-called *subprotocols*. Subprotocols are formalised by a preorder $\mathit{prot}_1 \sqsubseteq \mathit{prot}_2$, which captures (among others) a notion of swapping sends ahead of receives (provided that the send does not depend on the logical variables of the receive). We can then prove that prog_7 results in 42 by picking the following dependent separation protocols:

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

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

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

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

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

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

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

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

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

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

3.1.4 Formal Correspondence to Session Types

Even though Actris's notion of dependent separation protocols is influenced by binary session types, this paper does not provide a formal correspondence between the two systems. However, since Actris is built on top of Iris, it forms a suitable foundation for building logical relation models of type systems. In related work by Hinrichsen et al. [2021c], Actris has been used to define a logical relations model of binary session types, with support for various forms of polymorphism and recursion, asynchronous subtyping, references, and locks/mutexes. Similar to the RustBelt project [Jung et al.

2018a], that work gives rise to an extensible approach for proving type safety, which can be used to manually prove the typing judgements of racy, but safe, programs that cannot be type checked using only the rules of the type system.

3.1.5 Contributions and Outline

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

- We introduce *dependent separation protocols* inspired by affine binary session types to model the transfer of resources (including higher-order functions) between channel endpoints. We show that they can be used to handle choice, recursion, and delegation (Section 3.2).
- We introduce *subprotocols* inspired by asynchronous session subtyping. This notion relaxes duality, allowing channels to send messages before receiving others, and gives rise to a more extensional approach to reasoning about dependent separation protocols, providing more flexibility in the design and reuse of protocols. We moreover show how Löb induction is used to reason about recursive subprotocols (Section 3.3).
- We demonstrate the benefits obtained from building Actris on top of Iris by showing how Iris's support for ghost state and locks can be used to prove functional correctness of programs using manifest sharing, *i.e.,* channel endpoints shared by multiple parties (Section 3.4).
- We provide a case study on Actris and its mechanisation in Coq by proving functional correctness of a variant of the map-reduce model by Dean and Ghemawat [2004] (Section 3.5).
- We give a model of dependent separation protocols in the Iris framework to prove safety (*i.e.,* session fidelity) and postcondition validity of our Hoare triples (Section 3.6).
- We provide a full mechanisation of Actris [Hinrichsen et al. 2021b] using the interactive theorem prover Coq. On top of our Coq mechanisation, we provide custom tactics, which we use to mechanise all examples in the paper (Section 3.7).

3.1.6 Differences from the Conference Version

This paper is an extension of the paper "Actris: Session-type based reasoning in separation logic" presented at the POPL'20 conference [Hinrichsen et al. 2020]. In this paper we present Actris 2.0, which extends Actris 1.0 with the notion of subprotocols. This extension introduces new logical connectives and proof rules, but also involves a significant overhaul of the original model and its Coq mechanisation. We additionally extend the model and mechanisation sections substantially, with additional details,

considerations, and examples, to give a better understanding of how Actris works and how it can be used. Concretely, this paper includes the following extensions compared to the conference version:

- An overview of subprotocols in the introduction (Section 3.1.3).
- A new section on Actris 2.0's notion of subprotocols (Section 3.3).
- An updated and expanded description of the model of Actris in Iris (Section 3.6).
- An expanded section on the Coq mechanisation, with examples of mechanised proofs using the custom tactics for Actris that we have developed (Section 3.7).

3.2 A Tour of Actris

This section demonstrates the core features of Actris. We first introduce the language (Section 3.2.1) and the logic (Section 3.2.2). We then introduce and iteratively extend a simple distributed merge sort algorithm to demonstrate the main features of Actris (Section 3.2.3–Section 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).

3.2.1 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:

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

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 $\texttt{let}\ \_ := e_1\ \texttt{in}\ e_2$. Here, the underscore $\_$ is an anonymous binder, *i.e.,* an arbitrary variable that is fresh in the body of the binding expression.

The language includes the usual operations for heap manipulation. New references can be created using $\texttt{ref}\ e$, dereferenced using $!\,e$, and assigned to using $e_1 \leftarrow e_2$. Concurrency is supported via $\texttt{fork}\ \{e\}$, which spawns a new thread e that is executed in the background. The language also supports atomic operations like compare-and-set (CAS), which can be used to implement lock-free data structures and synchronisation primitives, but these are omitted from the syntax.

Message passing is performed over bidirectional channels, which are represented using pairs of buffers $(\vec{v}_1, \vec{v}_2)$ of unbounded size. The $\texttt{new\_chan}\ ()$ operation creates a new channel whose buffers are empty, and returns a tuple of endpoints (c_1, c_2). Bidirectionality is obtained by having one endpoint receive from the others send

buffer and *vice versa*. That means, $\texttt{send}\ c_i\ v$ enqueues the value v in its own buffer, *i.e.,* $\vec{v}_i$, and $\texttt{recv}\ c_i$ dequeues a value from the other buffer, *i.e.,* from $\vec{v}_2$ if $i = 1$ and from $\vec{v}_1$ if $i = 2$. Message passing is asynchronous, meaning that $\texttt{send}\ c\ v$ will always reduce, while $\texttt{recv}\ c$ will block as long as the receiving buffer is empty. The exact semantics of the channels will be detailed in Section 3.6.5.

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

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

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

3.2.2 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 3.6, Actris is built as a library on top of the Iris framework [Jung et al. 2015; Krebbers et al. 2017a; Jung et al. 2016, 2018b] and thus inherits all features of Iris. For the purpose of this section, no prior knowledge of Iris is expected as the majority 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 3.2.4) and to define recursive protocols (Section 3.2.6). An extensive overview of Iris can be found in [Jung et al. 2018b], and a tutorial-style introduction can be found in [Birkedal and Bizjak 2020].

The grammar of Actris and a selection of its rules are displayed in Figures 3.1 and 3.2. The Actris grammar includes the polymorphic lambda-calculus[1] with a number of primitive types and terms operating on these types. Most important is the type iProp of propositions and the type iProto of dependent separation protocols. The typing judgement is mostly standard and can be derived from the use of meta variables—we use the meta variables P and Q for propositions, the meta variable *prot* for protocols, the meta variable v for values, and the meta variables t and u for general terms of any type. Apart from that, there is the implicit side-condition that recursive predicates defined using the recursion operator $\mu x : \tau.\, t$ should be *guarded*. That means, the variable x should appear under a *contractive* term construct. As is usual in logics with guarded recursion [Nakano 2000], the later $\triangleright$ modality is contractive and is used to define recursive predicates. But moreover, as we will demonstrate in Section 3.2.6, the constructors $!\vec{x} : \vec{\tau}\,\langle v\rangle\{P\}.\, prot$ and $?\vec{x} : \vec{\tau}\,\langle v\rangle\{P\}.\, prot$ of dependent separation protocols are contractive in the tail argument *prot* to enable the construction of recursive protocols. The rule μ-UNFOLD says that $\mu x : \tau.\, t$ is in fact a fixpoint of t.

To express program specifications, Actris features Hoare triples $\{P\}\ e\ \{v.\, Q\}$, where P is the precondition and Q the postcondition. The binder v can be used

[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:

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

Ordinary affine separation logic:

Affine
$P * Q \Rrightarrow P$

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

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

Ht-fork
$$\frac{\{P\}\, e\, \{\mathsf{True}\}}{\{P\}\, \texttt{fork}\, \{e\}\, \{\mathsf{True}\}}$$

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

Recursion:

▷-intro
$P \Rrightarrow \triangleright P$

Löb
$(\triangleright P \Rrightarrow P) \Rrightarrow P$

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

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

Heap manipulation:

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

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

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

Figure 3.1: The grammar and a selection of rules of Actris.

Message passing:

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

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

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

Dependent separation protocol dual and append:

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

Figure 3.2: The rules of the Actris Dependent separation protocols.

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

The novel feature of Actris is its support for dependent separation protocols to reason about message-passing programs. This is done using the $c \rightarrowtail \mathit{prot}$ 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 ($\mathbf{!}$) or received ($\mathbf{?}$), the proposition P denotes the ownership that is transferred along the message, and prot denotes

the protocol that describes the subsequent messages. The logical variables $\vec{x}:\vec{\tau}$ can be used to bind variables in v, P, and $prot$. For example, $\mathbf{!}\,(b : \mathbb{B})\,(\ell : \mathsf{Loc})\,(i : \mathbb{N})\,\langle (b,\ell)\rangle\{\ell \mapsto i * 10 < i\}.\, prot$ expresses that a pair of a boolean and an integer reference whose value is at least 10 is sent. We often omit the proposition $\{P\}$, which simply means it is True.

Apart from the constructors for dependent separation protocols, Actris provides two primitive operations. The $\overline{prot}$ connective denotes the *dual* of a protocol. As with conventional session types, it transforms the protocol by changing all sends (**!**) into receives (**?**), and *vice versa*. Taking the dual twice thus results in the original protocol. The connective $prot_1 \cdot prot_2$ *appends* the protocols $prot_1$ and $prot_2$, which is achieved by substituting any **end** in $prot_1$ with $prot_2$. Finally, $prot_1 \sqsubseteq prot_2$ states that the protocol $prot_1$ is a *subprotocol* of $prot_2$. The subprotocol relation and its proof rules will be described in Section 3.3.

The rule Ht-new allow ascribing any protocol to newly created channels using `new_chan` (), obtaining ownership of $c \rightarrowtail prot$ and $c' \rightarrowtail \overline{prot}$ for the respective endpoints. The duality of the protocol guarantees that any receive (**?**) is matched with a send (**!**) by the dual endpoint, which is crucial for establishing safety (*i.e.,* session fidelity, see Section 3.6.7).

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

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

3.2.3 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 3.3.

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

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

Figure 3.3: A distributed merge sort algorithm (the code for `lmerge` and `lsplit` 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:

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

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:

$$l \overset{\text{list}}{\mapsto}_I \vec{x} \triangleq \begin{cases} l \mapsto \texttt{inl}\ () & \text{if } \vec{x} = [\,] \\ \exists v_1\ l_2.\, l \mapsto \texttt{inr}\ (v_1, l_2) * I\ x_1\ v_1 * l_2 \overset{\text{list}}{\mapsto}_I \vec{x}_2 & \text{if } \vec{x} = [x_1] \cdot \vec{x}_2 \end{cases}$$

The channel c adheres to the following dependent separation protocol:

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

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

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

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

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

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

Figure 3.4: A version of the sort service that receives the comparison function over the channel.

`sort_service` requires that c follows the dual. Second, the pre- and postcondition of `sort_service` are generalised to have an arbitrary protocol *prot* appended at the end. It is important to write specifications this way, so they can be embedded in other protocols. We will see examples of that in Section 3.2.6 and Section 3.2.7.

The proofs of these specifications follow from symbolic execution using the rules Ht-new, Ht-send, Ht-recv, and the standard separation logic rules.

3.2.4 Transferring Functions

The distributed `sort_service` from the previous section (Figure 3.3) is parametric on a comparison function. To demonstrate Actris's support for reasoning about functions transferred over channels, we verify the correctness of the program $\texttt{sort\_service}_{func}$ in Figure 3.4, which receives the comparison function over the channel instead of via a lambda abstraction. To verify this program, we extend the protocol sort_prot from Section 3.2.3 as follows:

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

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

$$\{c \rightarrowtail \overline{\mathsf{sort\_prot}_{func}} \cdot prot\}\ \texttt{sort\_service}_{func}\ c\ \{c \rightarrowtail prot\}$$

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

3.2.5 Choice

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

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

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

We let `left` := `true` and `right` := `false` to be used together with `select` for readability's sake. Due to the higher-order nature of Actris, the usual protocol specifications for choice from session types can be encoded as regular logical branching within the protocols:

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

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

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

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

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

3.2.6 Recursive Protocols

We will now use choice and recursion to verify the correctness of a sorting service that supports performing multiple sorting jobs in sequence. The code of the sorting service $\texttt{sort\_service}_{rec}$ and a possible client $\texttt{sort\_client}_{rec}$ are displayed in Figure 3.5. The service $\texttt{sort\_service}_{rec}$ contains a loop in which choice is used to either terminate the service, or to sort an individual list using the distributed merge sort algorithm `sort_service` from Section 3.2.3. The client $\texttt{sort\_client}_{rec}$ uses the service to sort a nested linked list l of linked lists. It performs this job by starting a single instance of the service at c, and then sequentially sends requests to sort each inner linked list l' in l. Finally, the client selects the terminating branch to end the communication with the service. A protocol for interacting with the sorting service can be defined as follows:

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

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

It is important to point out that—as is usual in logics with guarded recursion [Nakano 2000]—the variable x should appear under a *contractive* term construct in the body t of $\mu x : \tau.\, t$. In our protocol, the recursive variable *rec* appears under the

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

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

Figure 3.5: A recursive version of the sort service that can perform multiple jobs in sequence (the code for the function `liter`, which applies a function to each element of the list, is standard and has been elided).

argument of $\oplus$, which is defined in terms of $!\,\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\,prot$, which, similarly to $?\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\,prot$, is contractive in the tail protocol $prot$. The specifications of the service and the client are as follows:

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

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

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

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

3.2.7 Delegation

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

An example of a program that uses delegation is the $\mathtt{sort\_service}_{del}$ variant of the recursive sorting service in Figure 3.6, which allows multiple sorting jobs to be performed in parallel. To enable parallelism, it delegates a new channel c' to an inner sorting service for each sorting job.

The client $\mathtt{sort\_client}_{del}$ once again uses the sorting service to sort a nested linked list l of linked lists. The client starts a connection c to the new service, and

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

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

Figure 3.6: A recursive version of the sort service that uses delegation to perform multiple jobs in parallel (the code for the function `lcons`, which adds an element to the head of a list, has been elided).

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 `liter recv`).

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

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

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

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

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

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

3.2.8 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 $\mathtt{sort\_service}_{fg}$, as shown in Figure 3.7, requires the input list to be sent element by element, after which the service sends

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

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

```
merge_fg cmp c c1 c2 :=
  branch c1 with
   right ⇒ assert false
  | left  ⇒
    let x := recv c1 in
    merge_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
```

Figure 3.7: 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).

the sorted list back in the same fashion. We use choice to indicate whether the whole list has been sent (`right`) or another element remains to be sent (`left`).

The structure of $\texttt{sort\_service}_{fg}$ is somewhat similar to the coarse-grained merge-sort algorithm that we have seen before. The base cases of the empty or the singleton list are handled initially. This is achieved by waiting for at least two values before starting the recursive sub-services c_1 and c_2. In the base cases the values are sent back immediately, as they are trivially sorted. The inductive case is handled by starting two sub-services at c_1 and c_2 that are sent the two initially received elements, respectively, after which the $\texttt{split}_{fg}$ function is used to receive and forward the remaining values to the sub-services alternatingly. Once the `right` flag is received, the $\texttt{split}_{fg}$ function terminates, and the algorithm moves to the second phase in which the $\texttt{merge}_{fg}$ function merges the stream of values returned by the sub-services and forwards them to the parent service.

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

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

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

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

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

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

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

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

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

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

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

3.3 Subprotocols

This section describes **Actris 2.0**, which extends Actris 1.0—as presented in the conference version of this paper [Hinrichsen et al. 2020]—with *subprotocols*, inspired by asynchronous subtyping of session types. Subprotocols have two key features. First, they expose the asynchronous nature of channels in the Actris logic by relaxing the requirements of duality, thereby making it possible to prove functional correctness of a larger class of programs. Second, they give rise to a more extensional approach to reasoning about dependent separation protocols, as we can work up to the subprotocol relation and not equality, thereby providing more flexibility in the design and reuse of protocols.

We first introduce the subprotocol relation and its proof rules (Section 3.3.1). We then show how subprotocols can be employed to prove a mapper service, which handles requests one at a time, while its client may send multiple requests up front (Section 3.3.2). Next, we verify a list reversal service whose protocol involves a minimal specification of lists, which we then reuse through subprotocols to obtain a protocol with a more generic specification of lists (Section 3.3.3). Finally, we show that the subprotocol relation is coinductive, and thereby when combined with Löb induction, can be used to reason about recursive procotols (Section 3.3.4).

3.3.1 The Subprotocol Relation

The dependent separation protocol of a channel is picked upon channel creation (using the rule Ht-new), which then determines how the channel endpoints should interact. To ensure safe communication, Actris adapts the notion of duality from session types, which requires every send (**!**) of one endpoint to be paired with a receive (**?**) for the other endpoint, and *vice versa*. However, strictly working with a channel's protocol and its dual is more restrictive than necessary, as either endpoint is agnostic to some variance from the protocol by the other endpoint. We capture the flexibility of safe variances via a new notion—the *subprotocol relation*:

$$prot_1 \sqsubseteq prot_2$$

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

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

The subprotocol relation is inspired by asynchronous subtyping for session types [Mostrous et al. 2009; Mostrous and Yoshida 2015], which allows (1) sending subtypes (contravariance), (2) receiving supertypes (covariance), and (3) swapping sends ahead of receives. These variations are sound, as (1) the originally expected type that is to be sent can be derived from the subtype, (2) the originally expected type that is to be received can be derived from the supertype, and (3) sends do not block because

channels are buffered in both directions. These variances, along with the swapping property, can be generalised to dependent separation protocols using the following proof rules:

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

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

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

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

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

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

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

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

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

The key idea to obtain these features is to consider the subprotocol relation $prot_1 \sqsubseteq prot_2$ as a first-class logical connective. That is, $prot_1 \sqsubseteq prot_2$ is an Iris proposition that can be combined freely with the logical connectives of Iris and Actris (*e.g.,* separating conjunction, separating implication, higher-order quantification). Since $prot_1 \sqsubseteq prot_2$ is an Iris proposition, the proof rules are in fact (separating) implications in Iris. For readability, we use inference rules to denote each rule $(P_1 * \cdots * P_n) \mathrel{-\!\!*} Q$ as:

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

Logical variable manipulation and resource transfer:

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

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

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

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

Monotonicity and swapping:

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

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

Reflexivity and transitivity:

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

Dual and append:

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

Channel ownership:

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

Figure 3.8: The rules of Actris 2.0 for subprotocols.

The full set of rules for subprotocols is shown in Figure 3.8. The first four rules account for logical variable manipulation and resource transfer: Rules $\sqsubseteq$-SEND-OUT and $\sqsubseteq$-RECV-OUT generalise over the logical variables $\vec{x} : \vec{\tau}$ and transfer ownership of P out of the weaker sending protocol $!\,\vec{x} : \vec{\tau}\,\langle v\rangle\{P\}.\,prot$, and stronger receiving protocol $?\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\,prot$, respectively. Rule $\sqsubseteq$-SEND-IN weakens a sending protocol $!\,\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\,prot$ by instantiating the logical variables $\vec{x}{:}\vec{\tau}$ and transferring ownership of $P[\vec{t}/\vec{x}]$ into the protocol. Dually, the rule $\sqsubseteq$-RECV-IN strengthens a receiving protocol $?\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\,prot$ by instantiating the logical variables $\vec{x}{:}\vec{\tau}$ and transferring ownership of $P[\vec{t}/\vec{x}]$ into the protocol.

To demonstrate the intuition behind these rules consider the following proof of the subprotocol relation presented in Section 3.1.3, where we transfer ownership of $\ell_1' \mapsto 20$ into a protocol, while instantiating the logical variable ℓ_1 accordingly:

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

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

The rules for monotonicity ($\sqsubseteq$-SEND-MONO and $\sqsubseteq$-RECV-MONO) and swapping ($\sqsubseteq$-SWAP) in Figure 3.8 differ in two aspects from the rules for monotonicity ($\sqsubseteq$-SEND-MONO' and $\sqsubseteq$-RECV-MONO') and swapping ($\sqsubseteq$-SWAP') that we have seen in the beginning of this section. First, the actual rules only apply to protocols whose head does not have logical variables $\vec{x} : \vec{\tau}$ and resources P, *i.e.,* protocols of the shape $!\,\langle v\rangle.\,prot$ or $?\langle v\rangle.\,prot$, instead of those of the shape $!\,\vec{x} : \vec{\tau}\,\langle v\rangle\{P\}.\,prot$ or $?\vec{x} : \vec{\tau}\,\langle v\rangle\{P\}.\,prot$. While this restriction might seem to make the rules more restrictive, the more general rules for monotonicity ($\sqsubseteq$-SEND-MONO' and $\sqsubseteq$-RECV-MONO') and swapping ($\sqsubseteq$-SWAP') are derivable from these simpler rules. This is done using the rules for logical variable manipulation and resource transfer. Second, the actual rules for monotonicity have a later modality ($\triangleright$) in their premise. The later modality makes these rules stronger (by $\triangleright$-INTRO we have that P entails $\triangleright P$), and thereby internalizes its coinductive nature into the Actris logic so LÖB induction can be used to prove subprotocol relations for recursive protocols (Section 3.3.4).

The remaining rules in Figure 3.8 express that the subprotocol relation is reflexive ($\sqsubseteq$-REFL) and transitive ($\sqsubseteq$-TRANS), as well as that the dual operation is anti-monotone ($\sqsubseteq$-DUAL) and the append operation is monotone ($\sqsubseteq$-APPEND).

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

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

Here we extend the protocol $!\,\langle v\rangle\{P\}.\,?\langle w\rangle\{Q\}.\,prot$ with a so-called *frame* R, which describes resources that must be sent along with the originally expected resources P, and which are reacquired along with the resources Q that are sent back. The above subprotocol relation mimics the frame rule of separation logic (Ht-frame), which makes it possible to apply specifications while maintaining a *frame* of resources R:

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

The frame-like subprotocol relation is proven as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{Q * R \twoheadrightarrow\ ?\langle w\rangle.\,prot \sqsubseteq\ ?\langle w\rangle\{Q * R\}.\,prot}\ \sqsubseteq\text{-recv-in}}{R \twoheadrightarrow\ ?\langle w\rangle\{Q\}.\,prot \sqsubseteq\ ?\langle w\rangle\{Q * R\}.\,prot}\ \sqsubseteq\text{-recv-out}}{R \twoheadrightarrow\ !\,\langle v\rangle.\,?\langle w\rangle\{Q\}.\,prot \sqsubseteq\ !\,\langle v\rangle.\,?\langle w\rangle\{Q * R\}.\,prot}\ \sqsubseteq\text{-send-mono}}{P * R \twoheadrightarrow\ !\,\langle v\rangle\{P\}.\,?\langle w\rangle\{Q\}.\,prot \sqsubseteq\ !\,\langle v\rangle.\,?\langle w\rangle\{Q * R\}.\,prot}\ \sqsubseteq\text{-send-in}}{!\,\langle v\rangle\{P\}.\,?\langle w\rangle\{Q\}.\,prot \sqsubseteq\ !\,\langle v\rangle\{P * R\}.\,?\langle w\rangle\{Q * R\}.\,prot}\ \sqsubseteq\text{-send-out}$$

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

3.3.2 Swapping

Subprotocols make it possible to verify message-passing programs whose order of sends and receives does not match up w.r.t. duality. As an example of such a program, let us consider the mapper service and client in Figure 3.9. The service `mapper_service` is a loop, which iteratively receives an element, maps a function over that element, and sends the resulting value back. Conversely, the client `mapper_client` sends all of the elements of the list l up front, and only requests the mapped results back once all elements have been sent. Since the former interleaves the sends and receives, while the latter does not, the dependent separation protocols for the service and client cannot be dual of each other. However, the communication between the service and client is in fact safe as messages are buffered. We now show that using subprotocols we can prove that this is indeed the case. We define the protocol based on the interleaved communication:

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

The protocol is parameterised by representation predicates I_T and I_U that relate language-level values to elements of type T and U in the Iris/Actris logic, and a

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

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

Figure 3.9: A mapper service whose verification relies on swapping (the code for the functions `send_all` and `recvN` has been elided).

function $f : T \to U$ in Iris/Actris that specifies the behaviour of the language-level function f_v. The connection between f and f_v is formalised as:

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

Since `mapper_prot` describes an interleaved sequence of transactions, `mapper_service` can be readily verified against the protocol $\overline{\texttt{mapper\_prot}}$ using just the symbolic execution rules from Section 3.2. However, to verify `mapper_client` against the protocol `mapper_prot`, we need to weaken the protocol using Actris's rules for subprotocols. Given a list of n elements, the subprotocol relation (together with an intermediate step) that describes this weakening is:

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

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

n times $\sqsubseteq$-SWAP'

Both steps are proven by induction on n. In the first step, we unfold the recursive protocol n times using μ-UNFOLD and the derived rule $(prot_1 \oplus prot_2) \sqsubseteq \mathbf{!}\,\langle\texttt{left}\rangle.\, prot_1$ to weaken the choices. Recall from Section 3.2.5 that $\oplus$ is defined in terms of the send protocol ($\mathbf{!}$), allowing us to prove $(prot_1 \oplus prot_2) \sqsubseteq \mathbf{!}\,\langle\texttt{left}\rangle.\, prot_1$ using $\sqsubseteq$-SEND-OUT and $\sqsubseteq$-SEND-IN. The second step involves swapping all sends ahead of the receives using the rule $\sqsubseteq$-SWAP'.

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

Concretely, we prove the following specifications for the service and the client:

$$\left\{\begin{array}{l}\mathsf{f\_spec}\ I_T\ I_U\ f\ f_\mathsf{v} * \\ c \rightarrowtail \overline{\texttt{mapper\_prot}\ I_T\ I_U\ f} \cdot \mathit{prot}\end{array}\right\}\ \texttt{mapper\_service}\ f_\mathsf{v}\ c\ \{c \rightarrowtail \mathit{prot}\}$$

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

3.3.3 Minimal Protocols

An essential feature of separation logic is the ability to assign strong and minimal specifications to libraries, so that each library can be verified once against its specification, which in turn can be used to verify as many client programs as possible. To achieve a similar goal for message-passing programs we would like to assign strong and minimal protocols to services, so that each service can be verified once against its protocol, which in turn can be used to verify as many clients as possible. One of the key ingredients of separation logic to allow for such specifications is the frame rule (Ht-frame). In Section 3.3.1 we showed that subprotocols allow framing in protocols. In this section we give a detailed example of framing in protocols by considering the service `list_rev_service` in Figure 3.10, which receives a linked list, reverses it, and sends it back.

To specify this service, we could use a protocol similar to the sorting service in Section 3.2.3, defined in terms of the representation predicate $l \overset{\text{list}}{\mapsto}_{I_T} \vec{x}$ for linked lists:

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

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

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

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

$$l \overset{\text{list}}{\mapsto} \vec{v} \triangleq \begin{cases} l \mapsto \texttt{inl}\ () & \text{if } \vec{v} = [\,] \\ \exists l_2.\, l \mapsto \texttt{inr}\ (v_1, l_2) * l_2 \overset{\text{list}}{\mapsto} \vec{v}_2 & \text{if } \vec{v} = [v_1] \cdot \vec{v}_2 \end{cases}$$

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

$$l \overset{\text{list}}{\mapsto}_{I_T} \vec{x} \ \ast\ast\ (\exists \vec{v}.\, l \overset{\text{list}}{\mapsto} \vec{v} * \mathop{\ast}_{(x,v)\in(\vec{x},\vec{v})} I_T\ x\ v) \qquad \text{(list-rel)}$$

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

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

Figure 3.10: A list reversing service (the code for the function `lreverse` has been elided).

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

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

We first frame the interpretation predicates owned by the list $\ast_{(x,v)\in(\vec{x},\vec{v})}.I_T\ x\ v$, using an approach similar to the frame example in Section 3.3.1, and then use LIST-REL to combine it with $l \overset{\text{list}}{\mapsto} \vec{v}$ and $l \overset{\text{list}}{\mapsto} \mathsf{reverse}\ \vec{v}$ for the sending and receiving step, to turn them into $l \overset{\text{list}}{\mapsto}_{I_T} \vec{x}$ and $l \overset{\text{list}}{\mapsto}_{I_T} \mathsf{reverse}\ \vec{x}$, respectively. Note that the logical variable $\vec{v}$ is changed into $\vec{x}$, using the subprotocol rules for logical variable manipulation. With this subprotocol relation at hand, it is possible to prove the following specifications for the service and client:

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

3.3.4 Subprotocols and Recursion

We conclude this section by showing how subprotocol relations involving recursive protocols can be proved using LÖB induction. Recall from Section 3.2 that the principle of LÖB induction is as follows:

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

By letting P to be $prot_1 \sqsubseteq prot_2$, this means that we can prove $prot_1 \sqsubseteq prot_2$ under the assumption of the induction hypothesis $\triangleright(prot_1 \sqsubseteq prot_2)$. The later modality ($\triangleright$)

ensures that we do not immediately use the induction hypothesis, but first apply the monotonicity rule for send ($\sqsubseteq$-send-mono) or receive ($\sqsubseteq$-recv-mono), which is done typically after unfolding the recursion operator using μ-unfold. The monotonicity rules $\sqsubseteq$-send-mono or $\sqsubseteq$-recv-mono contain a later modality ($\triangleright$) in their premise, which makes it possible to strip off the later of the induction hypotheses (by monotonicity of $\triangleright$).

Our approach for proving subprotocol relations using Löb induction is similar to that of Brandt and Henglein [1998] for proving subtyping relations for recursive types using coinduction. Brandt and Henglein [1998] however have a syntactic restriction on proofs to ensure that the induction hypothesis is not used immediately (*i.e.,* is used in a *contractive* fashion), while we use the later modality ($\triangleright$) of Iris to achieve that. To demonstrate how our approach works, we prove $prot_1 \sqsubseteq prot_2$, where:

$$prot_1 \triangleq \mu(rec : \mathsf{iProto}).\, (\mathsf{list\_rev\_prot} \cdot rec) \oplus \mathbf{end}$$
$$prot_2 \triangleq \mu(rec : \mathsf{iProto}).\, (\mathsf{list\_rev\_prot}_{I_T} \cdot rec) \oplus \mathbf{end}$$

Here, $\mathsf{list\_rev\_prot}$ and $\mathsf{list\_rev\_prot}_{I_T}$ are the protocols from Section 3.3.3, for which we already proved $\mathsf{list\_rev\_prot} \sqsubseteq \mathsf{list\_rev\_prot}_{I_T}$. The proof is as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{prot_1 \sqsubseteq prot_2 \twoheadrightarrow prot_1 \sqsubseteq prot_2}}{prot_1 \sqsubseteq prot_2 \twoheadrightarrow \mathsf{list\_rev\_prot} \cdot prot_1 \sqsubseteq \mathsf{list\_rev\_prot}_{I_T} \cdot prot_2}\,(*)}{\triangleright(prot_1 \sqsubseteq prot_2) \twoheadrightarrow \triangleright(\mathsf{list\_rev\_prot} \cdot prot_1 \sqsubseteq \mathsf{list\_rev\_prot}_{I_T} \cdot prot_2)}\,\triangleright\text{ mono}}{\triangleright(prot_1 \sqsubseteq prot_2) \twoheadrightarrow (\mathsf{list\_rev\_prot} \cdot prot_1) \oplus \mathbf{end} \sqsubseteq (\mathsf{list\_rev\_prot}_{I_T} \cdot prot_2) \oplus \mathbf{end}}\,\oplus\text{ mono}}{\triangleright(prot_1 \sqsubseteq prot_2) \twoheadrightarrow prot_1 \sqsubseteq prot_2}\,\mu\text{-unfold}}{prot_1 \sqsubseteq prot_2}\,\text{Löb}$$

The proof starts with rule Löb and unfolding the recursive types. We then proceed with monotonicity of $\oplus$, *i.e.,* $\triangleright(prot_1 \sqsubseteq prot_2 \land prot_3 \sqsubseteq prot_4) \twoheadrightarrow (prot_1 \oplus prot_3) \sqsubseteq (prot_2 \oplus prot_4)$, which itself follows from $\sqsubseteq$-send-mono since selection ($\oplus$) is defined in terms of send (**!**). In step ($*$), we use $\sqsubseteq$-append and $\mathsf{list\_rev\_prot} \sqsubseteq \mathsf{list\_rev\_prot}_{I_T}$, which we proved in Section 3.3.3.

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

$$prot_1 \triangleq \mu(rec : \mathsf{iProto}).\, \mathbf{!}\,(x : \mathbb{Z})\,\langle x\rangle.\, \mathbf{?}\langle x+2\rangle.\, rec$$
$$prot_2 \triangleq \mu(rec : \mathsf{iProto}).\, \mathbf{!}\,(x : \mathbb{Z})\,\langle x\rangle.\, \mathbf{!}\,(y : \mathbb{Z})\,\langle y\rangle.\, \mathbf{?}\langle x+2\rangle.\, \mathbf{?}\langle y+2\rangle.\, rec$$

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

The proof is as follows:

$$\dfrac{}{prot_1 \sqsubseteq prot_2 \mathrel{-\!\!*} prot_1 \sqsubseteq prot_2}\ \sqsubseteq\text{-RECV-MONO}$$

$$\dfrac{prot_1 \sqsubseteq prot_2 \mathrel{-\!\!*} prot_1 \sqsubseteq prot_2}{prot_1 \sqsubseteq prot_2 \mathrel{-\!\!*} \mathsf{?}\langle x{+}2\rangle.\,\mathsf{?}\langle y{+}2\rangle.\,prot_1 \sqsubseteq \mathsf{?}\langle x{+}2\rangle.\,\mathsf{?}\langle y{+}2\rangle.\,prot_2}\ \sqsubseteq\text{-SEND-MONO'}$$

$$\dfrac{prot_1 \sqsubseteq prot_2 \mathrel{-\!\!*} \mathsf{?}\langle x{+}2\rangle.\,\mathsf{?}\langle y{+}2\rangle.\,prot_1 \sqsubseteq \mathsf{?}\langle x{+}2\rangle.\,\mathsf{?}\langle y{+}2\rangle.\,prot_2}{prot_1 \sqsubseteq prot_2 \mathrel{-\!\!*} \mathsf{!}\,y\,\langle y\rangle.\,\mathsf{?}\langle x{+}2\rangle.\,\mathsf{?}\langle y{+}2\rangle.\,prot_1 \sqsubseteq \mathsf{!}\,y\,\langle y\rangle.\,\mathsf{?}\langle x{+}2\rangle.\,\mathsf{?}\langle y{+}2\rangle.\,prot_2}\ \sqsubseteq\text{-SWAP'}$$

$$\dfrac{prot_1 \sqsubseteq prot_2 \mathrel{-\!\!*} \mathsf{!}\,y\,\langle y\rangle.\,\mathsf{?}\langle x{+}2\rangle.\,\mathsf{?}\langle y{+}2\rangle.\,prot_1 \sqsubseteq \mathsf{!}\,y\,\langle y\rangle.\,\mathsf{?}\langle x{+}2\rangle.\,\mathsf{?}\langle y{+}2\rangle.\,prot_2}{prot_1 \sqsubseteq prot_2 \mathrel{-\!\!*} \mathsf{?}\langle x{+}2\rangle.\,\mathsf{!}\,y\,\langle y\rangle.\,\mathsf{?}\langle y{+}2\rangle.\,prot_1 \sqsubseteq \mathsf{?}\langle x{+}2\rangle.\,\mathsf{!}\,y\,\langle y\rangle.\,\mathsf{?}\langle y{+}2\rangle.\,prot_2}\ \triangleright\text{-mono}$$

$$\dfrac{\triangleright(prot_1 \sqsubseteq prot_2) \mathrel{-\!\!*} \mathsf{?}\langle x{+}2\rangle.\,\mathsf{!}\,y\,\langle y\rangle.\,\mathsf{?}\langle y{+}2\rangle.\,prot_1 \sqsubseteq \mathsf{!}\,y\,\langle y\rangle.\,\mathsf{?}\langle x{+}2\rangle.\,\mathsf{?}\langle y{+}2\rangle.\,prot_2}{\triangleright(prot_1 \sqsubseteq prot_2) \mathrel{-\!\!*} \triangleright(\mathsf{?}\langle x{+}2\rangle.\,\mathsf{!}\,y\,\langle y\rangle.\,\mathsf{?}\langle y{+}2\rangle.\,prot_1 \sqsubseteq \mathsf{!}\,y\,\langle y\rangle.\,\mathsf{?}\langle x{+}2\rangle.\,\mathsf{?}\langle y{+}2\rangle.\,prot_2)}\ \sqsubseteq\text{-SEND-MONO'}$$

$$\dfrac{\triangleright(prot_1 \sqsubseteq prot_2) \mathrel{-\!\!*} \mathsf{!}\,x\,\langle x\rangle.\,\mathsf{?}\langle x{+}2\rangle.\,\mathsf{!}\,y\,\langle y\rangle.\,\mathsf{?}\langle y{+}2\rangle.\,prot_1 \sqsubseteq \mathsf{!}\,x\,\langle x\rangle.\,\mathsf{!}\,y\,\langle y\rangle.\,\mathsf{?}\langle x{+}2\rangle.\,\mathsf{?}\langle y{+}2\rangle.\,prot_2}{\triangleright(prot_1 \sqsubseteq prot_2) \mathrel{-\!\!*} \mathsf{?}\langle x{+}2\rangle.\,\mathsf{!}\,y\,\langle y\rangle.\,\mathsf{?}\langle x{+}2\rangle.\,\mathsf{?}\langle y{+}2\rangle.\,prot_1 \sqsubseteq \mathsf{?}\langle x{+}2\rangle.\,\mathsf{!}\,y\,\langle y\rangle.\,\mathsf{?}\langle x{+}2\rangle.\,\mathsf{?}\langle y{+}2\rangle.\,prot_2}\ \mu\text{-UNFOLD}$$

$$\dfrac{\triangleright(prot_1 \sqsubseteq prot_2) \mathrel{-\!\!*} prot_1 \sqsubseteq prot_2}{prot_1 \sqsubseteq prot_2}\ \text{LÖB}$$

After we use $\sqsubseteq$-SEND-MONO' for the first time, we strip off the later of the induction hypothesis $\triangleright(prot_1 \sqsubseteq prot_2)$. Subsequently, when we use $\sqsubseteq$-SEND-MONO' and $\sqsubseteq$-RECV-MONO, there are no more laters to strip. We therefore introduce the later implicitly using $\triangleright$-INTRO after applying the appropriate monotonicity rule (not shown).

3.4 Manifest Sharing via Locks

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

In this section we demonstrate how dependent separation protocols can be combined with lock-based concurrency. This combination allows us to prove functional correctness of programs that make use of the notion of *manifest sharing* [Balzer and Pfenning 2017; Balzer et al. 2019], where channel endpoints are shared between multiple parties. Instead of having to extend Actris, we make use of locks and ghost state that Actris inherits from Iris. We present the basic idea with an introductory example of sharing a channel endpoint between two parties (Section 3.4.1). We then consider a more challenging example of a distributed load-balancing mapper (Section 3.4.2).

3.4.1 Locks and Ghost State

Using the language from Section 3.2.1 it is possible to implement locks using a spin lock, ticket lock, or a more sophisticated implementation. For the purpose of this

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

Figure 3.11: The rules Actris inherits from Iris for locks.

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

Figure 3.12: A sample program that combines locks and channels to achieve manifest sharing.

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 3.11.

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 3.11 make use of the representation predicate $\mathsf{is\_lock}\ lk\ R$, which expresses that a lock lk guards the resources described by the proposition R. When creating a new lock one has to give up ownership of R, and in turn, obtains the representation predicate $\mathsf{is\_lock}\ lk\ R$ (Ht-new-lock). The representation predicate can then be freely duplicated so it can be shared between multiple threads (Lock-dup). When entering a critical section using `acquire` lk, a thread gets exclusive ownership of R (Ht-acquire), which has to be 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 3.12, which uses a lock to share a channel endpoint between two threads, that each send 21 to the main thread. The following dependent protocol, where n denotes the number of messages that should be exchanged, captures the expected interaction from the point of view of the main thread:

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

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:

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

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

Figure 3.13: 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. 2015, 2018b].

The $\mathsf{auth}_\gamma\, n$ fragment can be thought of as an authority that keeps track of the number of ongoing contributions n, while each $\mathsf{contrib}_\gamma$ is a token that witnesses that a contribution is still in progress. These concepts are made precise by the rules in Figure 3.13. 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 3.13 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:

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

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

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

To establish the initial lock invariant, we use the rules Auth-init and Auth-alloc to create the authority $\mathsf{auth}_\gamma\, 2$ and two $\mathsf{contrib}_\gamma$ tokens. The $\mathsf{contrib}_\gamma$ tokens play a crucial role in the proofs of the sending threads to establish that the existentially quantified variable n is positive (using Auth-contrib-pos). Knowing $n > 0$, these

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

Figure 3.14: A worker of the distributed mapper service.

threads can establish that the protocol $\overline{\texttt{lock\_prot}\ n}$ has not terminated yet (*i.e.,* is not **end**). This is needed to use the rule Ht-send to prove the correctness of sending 21, and thereby advancing the protocol from $\overline{\texttt{lock\_prot}\ n}$ to $\overline{\texttt{lock\_prot}\ (n-1)}$. Subsequently, the sending threads can deallocate the token $\mathsf{contrib}_\gamma$ (using Auth-dealloc) to decrement the n of $\mathsf{auth}_\gamma\ n$ accordingly to restore the lock invariant.

3.4.2 A Distributed Load-Balancing Mapper

This section demonstrates a more interesting use of manifest sharing. We show how Actris can be used to verify functional correctness of a distributed load-balancing mapper that maps a function f_v over a list. Our distributed mapper consists of one client that distributes the work, and a number of workers that perform the function f_v on individual elements of the list. To enable communication between the client and the workers, we make use of a single channel. One endpoint is used by the client to distribute the work between the workers, while the other endpoint is shared between all workers to request and return work from the client. The implementation of the workers, which can be found in Figure 3.14, consists of a loop over three phases:

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

The first and last phases are in a critical section guarded by a lock lk since they involve interaction over a shared channel endpoint. As the sharing behaviour is encapsulated by the worker, we omit the code of the client for brevity's sake.[2]

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

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

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

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

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

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

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

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

Figure 3.15: The authoritative contribution ghost theory extended with multisets.

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

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

Similarly to `mapper_prot` from Section 3.3.2, the protocol is parameterised by representation predicates I_T and I_U, and a function $f : T \to \mathsf{List}\ U$ in Iris/Actris logic, related through the `f_spec` specification. Similar to the protocol `lock_prot` from Section 3.4.1, the protocol par_mapper_prot is indexed by the number of remaining workers n. On top of that, it carries a multiset X describing the values currently being processed by all the workers. The multiset X is used to make sure that the returned results are in fact the result of mapping the function f. The condition $(n = 1) \Rightarrow (X = \emptyset)$ on the branching operator $(\&)$ expresses that the last worker may only request more work if there are no ongoing jobs.

To accommodate sharing of the channel endpoint between all workers using a lock invariant, we extend the authoritative contribution ghost theory from Section 3.4.1. We do this by adding multisets X and Y to the connectives $\mathsf{auth}_\gamma\, n\, X$ and $\mathsf{contrib}_\gamma\, Y$. These multisets keep track of the values held by the workers. The rules for the ghost theory extended with multisets are shown in Figure 3.15. The rules AuthM-init, AuthM-alloc and AuthM-dealloc are straightforward generalisations of the ones we have seen before. The new rules AuthM-add and AuthM-remove determine that the multiset Y of $\mathsf{contrib}_\gamma\, Y$ can be updated as long as it is done in accordance with the multiset X of $\mathsf{auth}_\gamma\, n\, X$. Finally, the AuthM-contrib-agree rule expresses that the multiset Y of $\mathsf{contrib}_\gamma\, Y$ must be a subset of the multiset X of $\mathsf{auth}_\gamma\, n\, X$, while the stricter rule AuthM-contrib-agree1 asserts equality between X and Y when only one contribution remains.

The specifications of `par_mapper_worker` and a top-level client `par_mapper_client` that uses n workers to map f_v over the linked list ℓ are as follows:

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

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

The lock invariant and specification of `par_mapper_worker` are similar to those used in the simple example in Section 3.4.1. The specification of `par_mapper_client` $n\ f_\mathsf{v}\ \ell$ simply states that the resulting linked list points to a permutation of performing the map at the level of the logic. To specify that, we make use of $\texttt{flatMap} : (T \to \mathsf{List}\ U) \to (\mathsf{List}\ T \to \mathsf{List}\ U)$, whose definition is standard.

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

3.5 Case Study: Map-Reduce

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

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

In order to implement and verify our map-reduce version we make use of the implementation and verification of the fine-grained distributed merge sort algorithm (Section 3.2.8) and the distributed load-balancing mapper (Section 3.4.2). As such, our map-reduce implementation is mostly a suitable client that glues together communication with these services. The purpose of this section is to give a high-level description of the implementation. The actual code and proofs can be found in the accompanied Coq development [Hinrichsen et al. 2021b].

3.5.1 A Functional Specification of Map-Reduce

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

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

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

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

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

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

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

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

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

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

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

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

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

3.5.2 Implementation of Map-Reduce

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

1. Split the input data into chunks and delegate these chunks to the mapper nodes, that each apply the "map" function f to their given data in parallel.
2. Collect the complete set of mapped results and "shuffle" them, *i.e.,* group them by key. The grouping is often implemented with 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 $\texttt{map\_reduce}_\mathsf{v}\ n\ m\ f_\mathsf{v}\ g_\mathsf{v}\ l$, which coordinates the work for performing map-reduce on a linked list l between n mappers performing the "map" function f_v and m workers performing the "reduce" function g_v. To make the implementation more interesting, we prevent storing intermediate values locally by forwarding/returning them immediately as they are available/requested. The global structure is as follows:

1. Start n instances of the load-balancing `par_mapper_worker` from Section 3.4 with the f_v function. Additionally start an instance of $\texttt{sort\_service}_{fg}$ from Section 3.2, parameterised by a concrete comparison function on the keys, corresponding to $\lambda(k_1, \_)\ (k_2, \_).\ k_1 < k_2$. Note that the type of keys are restricted to be $\mathbb{Z}$ for brevity's sake.
2. Perform a loop that handles communication with the mappers. If a mapper requests work, pop a value from the input list. If a mapper returns work, forward it to the sorting service. This process is repeated until all inputs have been mapped and forwarded.
3. Start m instances of the `par_mapper_worker`, parameterised by g_v.
4. Perform a loop that handles communication with the mappers. If a mapper requests work, group elements returned by the sort service. If a mapper returns work, aggregate the returned value in a the linked list. Grouped elements are created by requesting and aggregating elements from the sorter until the key changes.

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

3.5.3 Functional Correctness of Map-Reduce

The specification of the map-reduce program is as follows:

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

The `f_spec` predicates (as introduced in Section 3.3.2) establish a connection between the functions f and g on the logical level and the functions f_v and g_v in the language. These make use of the various interpretation predicates I_T, $I_{\mathbb{Z}*U}$, $I_{\mathbb{Z}*\mathsf{List}\ U}$, and I_V for the types in question. Lastly, the $l \overset{\text{list}}{\mapsto}_{I_T} \vec{x}$ predicate determines that the input is a linked list of the initial type T. The postcondition asserts that the result $\vec{z}$ is a permutation of the original linked list $\vec{x}$ applied to the functional specification `map_reduce` of map-reduce from Section 3.5.1.

3.6 The Model of Actris

We prove the adequacy theorem of Actris—given a Hoare triple $\{\mathsf{True}\}\, e\, \{\phi\}$ that is derivable in Actris, we prove that e cannot get stuck (*i.e.,* safety), and if e terminates, the resulting value v satisfies ϕ (*i.e.,* postcondition validity). To do that, we construct a model of Actris as a shallow embedding in the Iris framework [Jung et al. 2015; Krebbers et al. 2017a; Jung et al. 2016, 2018b]. This means that the type iProto of dependent separation protocols, the subprotocol relation $prot_1 \sqsubseteq prot_2$, and the connective $c \rightarrowtail prot$ for the channel ownership, are definitions in Iris, and the Actris proof rules are lemmas about these definitions in Iris. Adequacy of Actris is then simply a consequence of Iris's adequacy theorem.

In this section we describe the relevant aspects of the model of Actris. We model the type iProto of dependent separation protocols as the solution of a recursive domain equation, and describe how the operators for dual and composition are defined (Section 3.6.1). We then define the subprotocol relation $prot_1 \sqsubseteq prot_2$ and prove its proof rules as lemmas (Section 3.6.2). To connect protocols to the endpoint channel buffers in the semantics we define the *protocol consistency relation*, which ensures that a pair of protocols is consistent with the messages in their associated buffers (Section 3.6.3). On top of the protocol consistency relation, we define the *Actris ghost theory* for dependent separation protocols (Section 3.6.4), which forms the key ingredient for defining the connective $c \rightarrowtail prot$ for channel ownership (Section 3.6.6) that links protocols to the semantics of channels (Section 3.6.5). We then show how adequacy follows from the embedding in Iris (Section 3.6.7). Finally, we show how to solve the recursive domain equation for the type iProto of dependent separation protocols (Section 3.6.8).

3.6.1 The Model of Dependent Separation Protocols

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

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

The left part of the sum type (the unit type 1) indicates that the protocol has terminated, while the right part describes a message that is exchanged, expressed as an Iris predicate. Since the recursive occurrence of iProto in the predicate appears in negative position, we guard it using Iris's *type-level later* ($\blacktriangleright$) operator (whose only constructor is $\mathsf{next} : T \to \blacktriangleright\, T$).

The exact way the solution is constructed is detailed in Section 3.6.8. For now, we assume a solution exists, and define the dependent separation protocols constructors as follows:

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

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

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

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

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

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

$$\begin{aligned}
\overline{\mathbf{send}} &\triangleq \mathbf{recv} \\
\overline{\mathbf{recv}} &\triangleq \mathbf{send} \\
\overline{(_)} &\triangleq \mu rec.\, \lambda prot.\, \begin{cases} \mathtt{inj}_1\,() & \text{if } prot = \mathtt{inj}_1\,() \\ \mathtt{inj}_2\,(\overline{a}, \lambda w\ prot'.\, \exists prot''. \\ \qquad \Phi\ w\ (\mathsf{next}\ prot'') * \\ \qquad prot' = \mathsf{next}\ (rec\ prot'')) & \text{if } prot = \mathtt{inj}_2\,(a, \Phi) \end{cases} \\
(_ \cdot prot_2) &\triangleq \mu rec.\, \lambda prot_1.\, \begin{cases} prot_2 & \text{if } prot_1 = \mathtt{inj}_1\,() \\ \mathtt{inj}_2\,(a, \lambda w\ prot'.\, \exists prot''. \\ \qquad \Phi\ w\ (\mathsf{next}\ prot'') * \\ \qquad prot' = \mathsf{next}\ (rec\ prot'')) & \text{if } prot_1 = \mathtt{inj}_2\,(a, \Phi) \end{cases}
\end{aligned}$$

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

The equational rules for dual $\overline{(\_)}$ and append $(\_ \cdot \_)$ from Figure 3.1 are proven as lemmas in Iris using Löb induction. This is possible as the recursive call *rec prot″* appears below a next constructor—since the next constructor is contractive, we can strip-off the later from the induction hypothesis when proving the equality for the tail.

Difference from the Conference Version In the conference version of this paper [Hinrichsen et al. 2020], we described two versions of the recursive domain equation for dependent separation protocols: an "ideal" version (as used in this paper), where iProto appears in negative position, and an "alternative" version, where iProto appears in positive position. At that time, we were unable to construct a solution of the "ideal" version, so we used the "alternative" version. In Section 3.6.8 we show how we are now able to solve the "ideal" version.

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

3.6.2 The Model of the Subprotocol Relation

We now model the subprotocol relation $prot_1 \sqsubseteq prot_2$ from Section 3.3. For legibility, we present it in the style of an inference system through its constructors, whereas it is formally defined using Iris's guarded recursion operator $(\mu x.\,t)$:

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

$$\frac{\forall v, prot_2.\ \Phi_2\ v\ (\mathsf{next}\ prot_2) \mathrel{-\!\!*} \exists prot_1.\ \Phi_1\ v\ (\mathsf{next}\ prot_1) * \triangleright(prot_1 \sqsubseteq prot_2)}{\mathtt{inj}_2\,(\mathbf{send}, \Phi_1) \sqsubseteq \mathtt{inj}_2\,(\mathbf{send}, \Phi_2)} \qquad \frac{\forall v, prot_1.\ \Phi_1\ v\ (\mathsf{next}\ prot_1) \mathrel{-\!\!*} \exists prot_2.\ \Phi_2\ v\ (\mathsf{next}\ prot_2) * \triangleright(prot_1 \sqsubseteq prot_2)}{\mathtt{inj}_2\,(\mathbf{recv}, \Phi_1) \sqsubseteq \mathtt{inj}_2\,(\mathbf{recv}, \Phi_2)}$$

$$\frac{\forall v_1, v_2, prot_1, prot_2.\ (\Phi_1\ v_1\ (\mathsf{next}\ prot_1) * \Phi_2\ v_2\ (\mathsf{next}\ prot_2)) \mathrel{-\!\!*} \exists prot.\ \triangleright(prot_1 \sqsubseteq\ !\,\langle v_2\rangle.\,prot) * \triangleright(?\langle v_1\rangle.\,prot \sqsubseteq prot_2)}{\mathtt{inj}_2\,(\mathbf{recv}, \Phi_1) \sqsubseteq \mathtt{inj}_2\,(\mathbf{send}, \Phi_2)}$$

To be a well-formed guarded recursion definition, every recursive occurrence of $\sqsubseteq$ is guarded by later modality ($\triangleright$). Aside from the later being required for well-formedness, these laters make it possible to reason about the subprotocol relation using Löb induction; both to prove the subprotocol rules from Figure 3.8 as lemmas, and for Actris users to reason about recursive protocols as shown in Section 3.3.4. The relation is defined in a syntax-directed fashion (*i.e.,* there are no overlapping rules), and therefore all constructors need to be defined so that they are closed under monotonicity and transitivity.

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

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

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

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

To obtain syntax-directed rules, the first rule combines $\sqsubseteq$-send-out, $\sqsubseteq$-send-in, and $\sqsubseteq$-send-mono, and dually, the second rule combines $\sqsubseteq$-recv-out, $\sqsubseteq$-recv-in, and $\sqsubseteq$-recv-mono. The third rule combines $\sqsubseteq$-recv-out, $\sqsubseteq$-send-out and $\sqsubseteq$-swap and bakes in transitivity, instead of asserting that $prot_1$ and $prot_2$ are equal to $!\,\langle v_2\rangle.\ prot$ and $?\langle v_1\rangle.\ prot$, respectively.

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

The rules in Figure 3.8 on page 62 are proven as lemmas. Those for logical variable and resource manipulation ($\sqsubseteq$-send-out, $\sqsubseteq$-send-in, $\sqsubseteq$-recv-out and $\sqsubseteq$-recv-in) monotonicity ($\sqsubseteq$-send-mono and $\sqsubseteq$-recv-mono), and swapping ($\sqsubseteq$-swap) follow almost immediately from the definition, whereas those for reflexivity ($\sqsubseteq$-refl), transitivity ($\sqsubseteq$-trans), and the dual and append operator ($\sqsubseteq$-dual and $\sqsubseteq$-append) are proven using Löb induction.

3.6.3 Protocol Consistency

To connect dependent separation protocols to the semantics of channels in Section 3.6.6, we define the *protocol consistency relation* $\mathsf{prot\_consistent}\ \vec{v}_1\ \vec{v}_2\ prot_1\ prot_2$, which expresses that protocols $prot_1$ and $prot_2$ are *consistent* w.r.t. channel buffers containing values $\vec{v}_1$ and $\vec{v}_2$. The consistency relation is defined as:

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

Intuitively, $\mathsf{prot\_consistent}\ \vec{v}_1\ \vec{v}_2\ prot_1\ prot_2$ ensures that for all messages $\vec{v}_1$ in transit from the endpoint described by $prot_1$ to the endpoint described by $prot_2$, the protocol

prot_2 is expecting to receive these message in order (and *vice versa* for $\vec{v}_2$), after which the remaining protocols prot and $\overline{\mathit{prot}}$ are dual. To account for weakening we close the relation under subprotocols (by using $\sqsubseteq$ instead of equality), which additionally captures ownership of the resources associated with the messages $\vec{v}_1$ and $\vec{v}_2$.

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

$$\mathsf{prot\_consistent}\ [\,]\ [\,]\ \mathit{prot}\ \overline{\mathit{prot}}$$

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

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

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

The last two rules are proved by inversion on the definition of the subprotocol relation ($\sqsubseteq$). Since the subprotocol relation ($\sqsubseteq$) is defined using guarded recursion, we obtain a later modality ($\triangleright$) for each inversion. To prove the first rule, we need to perform a number of inversions equal to the size of the buffer $\vec{v}_2$, whereas for the second rule we need to perform just a single inversion. The later modalities will be eliminated through physical program steps in the semantics of channels in Section 3.6.5.

3.6.4 The Actris Ghost Theory

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

We use an approach similar to the ghost theory for contributions that we used in Section 3.4. The authority $\mathsf{prot\_ctx}\ (\gamma_1,\gamma_2)\ \vec{v}_1\ \vec{v}_2$ governs the global state of the buffers $\vec{v}_1$ and $\vec{v}_2$. The tokens $\mathsf{prot\_own_l}\ (\gamma_1,\gamma_2)\ \mathit{prot}_l$ and $\mathsf{prot\_own_r}\ (\gamma_1,\gamma_2)\ \mathit{prot}_r$ provide local views of that state, being that the protocols prot_l and prot_r are consistent with the buffers. As we will see in Section 3.6.6, the authority can then be shared through a lock, while the tokens can be distributed to individual threads. The ghost connectives are identified by the shared ghost identifiers γ_1 and γ_2 for the protocols prot_l and prot_r, respectively.

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

Figure 3.16: Higher-order ghost variables in Iris.

To define the connectives of the Actris ghost theory we use Iris's higher-order ghost variables, whose rules are shown in Figure 3.16. Higher-order ghost variables come in pairs $\gamma \mapsto_\bullet prot$ and $\gamma \mapsto_\circ prot$, which always hold the same protocol $prot$. They can be allocated together (HO-GHOST-ALLOC), are always required to hold the same protocol (HO-GHOST-AGREE), and can only be updated together (HO-GHOST-UPDATE). The subtle part of the higher-order ghost variables is that they involve ownership of a protocol of type iProto, which is defined in terms of Iris propositions iProp. Due to the dependency on iProp, which is covered in detail in Section 3.6.8, the rule HO-GHOST-AGREE only gives the equality between the protocols under a later modality ($\triangleright$). The Actris ghost theory connectives are then defined as:

$$\mathsf{prot\_ctx}\ (\gamma_1,\gamma_2)\ \vec{v}_1\ \vec{v}_2 \triangleq \exists prot_1, prot_2.\, \gamma_1 \mapsto_\bullet prot_1 * \gamma_2 \mapsto_\bullet prot_2 * \triangleright\, \mathsf{prot\_consistent}\ \vec{v}_1\ \vec{v}_2\ prot_1\ prot_2$$
$$\mathsf{prot\_own_l}\ (\gamma_1,\gamma_2)\ prot_l \triangleq \exists prot_l'.\, \gamma_1 \mapsto_\circ prot_l' * \triangleright(prot_l' \sqsubseteq prot_l)$$
$$\mathsf{prot\_own_r}\ (\gamma_1,\gamma_2)\ prot_r \triangleq \exists prot_r'.\, \gamma_2 \mapsto_\circ prot_r' * \triangleright(prot_r' \sqsubseteq prot_r)$$

The authority $\mathsf{prot\_ctx}\ (\gamma_1,\gamma_2)\ \vec{v}_1\ \vec{v}_2$ asserts that the buffers $\vec{v}_1$ and $\vec{v}_2$ are consistent with respect to the protocols $prot_1$ and $prot_2$ (via $\mathsf{prot\_consistent}\ \vec{v}_1\ \vec{v}_2\ prot_1\ prot_2$). It also asserts the higher-order authoritative ownership $\gamma_1 \mapsto_\bullet prot_1$ and $\gamma_2 \mapsto_\bullet prot_2$ of both protocols. The tokens assert higher-order fragmented ownership $\gamma_1 \mapsto_\circ prot_l'$ and $\gamma_2 \mapsto_\circ prot_r'$ of protocols $prot_l'$ and $prot_r'$ that are weaker than the protocol arguments $prot_l$ and $prot_r$ (via $prot_l' \sqsubseteq prot_l$ and $prot_r' \sqsubseteq prot_r$). Explicit weakening under the subprotocol relation may seem redundant, as weakening is already accounted for in prot_consistent. However, it allows us to weaken the protocols of the tokens without the presence of the authority as shown by the rules PROTO-$\sqsubseteq$-L and PROTO-$\sqsubseteq$-R in Figure 3.17. The later modality ($\triangleright$) makes sure that $\mathsf{prot\_own_l}\ (\gamma_1,\gamma_2)\ prot$ and $\mathsf{prot\_own_r}\ (\gamma_1,\gamma_2)\ prot$ are contractive in $prot$. For readability we condense the ghost state identifiers (γ_1,γ_2) into a single identifier γ from now on.

With these definitions at hand, we prove the rules of the ghost theory presented in Figure 3.17. The rule PROTO-ALLOC corresponds to allocation of a buffer pair, the rules PROTO-SEND-L and PROTO-SEND-R correspond to sending a message, and the rules PROTO-RECV-L and PROTO-RECV-R correspond to receiving a message. These are proved through a combination of the rules for higher-order ghost state from Figure 3.16, and the rules for the protocol consistency relation prot_consistent from Section 3.6.3.

$$\mathsf{True} \Rrightarrow \exists \gamma.\, \mathsf{prot\_ctx}\ \gamma\ []\ []\ *\ \mathsf{prot\_own_l}\ \gamma\ \mathit{prot}\ *\ \mathsf{prot\_own_r}\ \gamma\ \overline{\mathit{prot}} \tag{PROTO-ALLOC}$$

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

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

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

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

$$\mathsf{prot\_own_l}\ \gamma\ \mathit{prot}\ *\ \mathit{prot} \sqsubseteq \mathit{prot}' \mathrel{-\!\!*} \mathsf{prot\_own_l}\ \gamma\ \mathit{prot}' \tag{PROTO-$\sqsubseteq$-L}$$

$$\mathsf{prot\_own_r}\ \gamma\ \mathit{prot}\ *\ \mathit{prot} \sqsubseteq \mathit{prot}' \mathrel{-\!\!*} \mathsf{prot\_own_r}\ \gamma\ \mathit{prot}' \tag{PROTO-$\sqsubseteq$-R}$$

Figure 3.17: The Actris ghost theory.

3.6.5 Semantics of Channels

Since Iris is parametric in the programming language that is used, there are various approaches to extend Iris with support for channels:

- Instantiate Iris with a language that has native support for channels. This approach was carried out in the original Iris paper [Jung et al. 2015] and by Tassarotti et al. [2017].
- Instantiate Iris with a language that has low-level concurrency primitives, but no native support for channels, and implement channels as a library in that language. This approach was carried out by Bizjak et al. [2019] for a lock-free implementation of channels.

In this paper we take the second approach. We use HeapLang, the default language shipped with Iris, and implement bidirectional channels using a pair of mutable linked lists protected by a lock. Although this implementation is not efficient, contrary to *e.g.,* the implementation by Bizjak et al. [2019], it has the benefit that it gives a clear declarative semantics that corresponds exactly to the intuitive semantics of channels described in Section 3.2.1.

Our implementation of bidirectional channels in HeapLang is displayed in Figure 3.18. New channels are created by the `new_chan` function, which allocates two empty mutable linked lists l and r using `lnil` (), along with a lock lk using `new_lock` (), and returns the tuples (l, r, lk) and (r, l, lk), where the order of the linked lists l and r determines the side of the endpoints. We refer to the list in the left position as the endpoint's own buffer, and the list in the right position as the other endpoint's buffer.

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

Figure 3.18: Implementation of bidirectional channels in HeapLang.

Values are sent over a channel endpoint (l, r, lk) using the `send` function. This function operates in an atomic fashion by first acquiring the lock via `acquire` lk, thereby entering the critical section, after which the value is enqueued (*i.e.,* appended to the end) of the endpoint's own buffer using `lsnoc` l v. The `skipN` $|r|$ instruction is a no-op that is inserted to aid the proof. We come back to the reason why this instruction is needed in Section 3.6.6.

Values are received over a channel endpoint (l, r, lk) using the `send` function, which performs a loop that repeatedly calls the helper function `try_recv`. This helper function attempts to receive a value atomically, and fails if there is no value in the other endpoint's buffer. The function `try_recv` acquires the lock with `acquire` lk, and then checks whether the other endpoint's buffer is empty using `lisnil` r. If it is empty, nothing is returned (*i.e.,* $\texttt{inj}_1\ ()$), while otherwise the value is dequeued and returned (*i.e.,* $\texttt{inj}_2\ (\texttt{lpop}\ l)$).

3.6.6 The Model of Channel Ownership

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

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

The predicate states that the referenced channel endpoint c is either the left (l, r, lk) or the right (r, l, lk) side of a channel, and that we have exclusive ownership of the ghost token $\mathsf{prot\_own_l}\ \gamma\ \mathit{prot}$ or $\mathsf{prot\_own_r}\ \gamma\ \mathit{prot}$ for the corresponding side. Iris's lock

representation predicate is_lock (previously presented in Section 3.4) is used to make sharing of the buffers possible. The lock invariant is governed by lock lk, and carries the ownership $l \overset{\text{list}}{\mapsto} \vec{v}_1$ and $r \overset{\text{list}}{\mapsto} \vec{v}_2$ of the mutable linked lists containing the channel buffers, as well as $\mathsf{prot\_ctx}\ \gamma\ \vec{v}_1\ \vec{v}_2$, which asserts protocol consistency of the buffers with respect to the protocols.

With the definition of the channel endpoint ownership along with the ghost theory and lock rules we then prove the channel rules HT-NEW, HT-SEND and HT-RECV from Figure 3.1. The proofs are carried out through symbolic execution to the point where the critical section is entered, after which the rules of the Actris ghost theory (Figure 3.17) are used to allocate or update the ghost state appropriately so that it matches the physical channel buffers.

The Need for Skip Instructions The rules PROTO-SEND-L and PROTO-SEND-R from Figure 3.17 contain a number of later modalities ($\triangleright$) proportional to the other endpoint's buffer in their premise. As explained in Section 3.6.3 these later modalities are the consequence of having to perform a number of inversions on the subprotocol relation, which is defined using guarded recursion, and thus contains a later modality for each recursive unfolding.

To eliminate these later modalities, we instrument the code of the `send` function with the `skipN` $|r|$ instruction, which performs a number of skips equal to the size of the other endpoint's buffer r. The `skipN` instruction has the following specification:

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

Instrumentation with skip instructions appears more often in work on step-indexing, see *e.g.,* [Svendsen et al. 2016; Giarrusso et al. 2020]. It is needed due to the fact that current step-indexed logics like Iris unify physical/program steps and logical steps, *i.e.,* for each physical/program step at most one later can be eliminated from the hypotheses. Svendsen et al. [2016] proposed a more liberal version of step-indexed, called *transfinite step-indexing*, to avoid this problem. However, transfinite step-indexing is not available in Iris.

3.6.7 Adequacy of Actris

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

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

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

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

3.6.8 Solving the Recursive Domain Equation for Protocols

Recall the recursive domain equation for dependent separation protocols from Section 3.6.1:

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

This recursive domain equation shows that iProto depends on the type iProp of Iris propositions. To use types that depend on iProp as part of higher-order ghost state in Iris, such types need to be bi-functorial in iProp. Hence, this means that to construct iProto, in a way that it can be used in combination with the higher-order ghost variables in Figure 3.16, we need to solve the following recursive domain equation:

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

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

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

$$\begin{aligned}&\mathsf{iProto}_2(X^-, X^+) \cong \\ &\quad 1 + (\mathsf{action} \times (\mathsf{Val} \to \blacktriangleright(1 + (\mathsf{action} \times (\mathsf{Val} \to \blacktriangleright\mathsf{iProto}_2(X^-, X^+) \to X^-))) \to X^+))\end{aligned}$$

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

3.7 Coq Mechanisation

The definition of the Actris logic, its model, and the proofs of all examples in this paper have been fully mechanised using the Coq proof assistant [Coq Development Team 2021]. In this section we will elaborate on the mechanisation effort (Section 3.7.1), and go through the full proof of a message-passing program (Section 3.7.2) and a subprotocol relation (Section 3.7.3) showcasing the tactics for Actris. During the section we display proofs and proof states taken directly from the Coq mechanisation, which differs in notation from the paper as shown in Figure 3.19.

3.7.1 Mechanisation Effort

The mechanisation of Actris is built on top of the mechanisation of Iris [Krebbers et al. 2017a; Jung et al. 2016, 2018b]. To carry out proofs in separation logic, we use the MoSeL Proof Mode (formerly Iris Proof Mode) [Krebbers et al. 2017b, 2018],

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

Figure 3.19: The difference in syntax between the paper and Coq source code.

which provides an embedded proof assistant for separation logic in Coq. Building Actris on top of the Iris and MoSeL framework in Coq has a number of tangible advantages:

- By defining channels on top of HeapLang, the default concurrent language shipped with Iris, we do not have to define a full programming language semantics, and can reuse all of the program libraries and Coq machinery, including the tactics for symbolic execution of non message-passing programs.
- Since Actris is mechanised as an Iris library that provides support for the iProto type, the subprotocol relation $prot_1 \sqsubseteq prot_2$, the $c \rightarrowtail prot$ connective, the various operations on protocols, and the proof rules as lemmas, we get all of the features of Iris for free, such as the ghost state mechanisms for reasoning about concurrency.
- When proving the Actris proof rules, we can make use of the MoSeL Proof Mode to carry out proofs directly using separation logic, thus reasoning at a high-level of abstraction.
- We can make use of the extendable nature of the MoSeL Proof Mode to define custom tactics for symbolic execution of message-passing programs.

These advantages made it possible to mechanise Actris, along with the examples of the paper, with a small Coq development of a total size of about 5000 lines of code (comments and whitespace included). The line count of the different components are shown in Figure 3.20.

3.7.2 Tactic Support for Session Type-Based Reasoning

To carry out interactive Actris's proof using symbolic execution, we follow the methodology described in the original Iris Proof Mode paper [Krebbers et al. 2017b], which in particular means that the logic in Coq is presented in weakest precondition style rather than using Hoare triples. For handling `send` or `recv` we define the following tactics:

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

These tactics roughly perform the following actions:

Component	Sections	∼LOC
The Actris model	Section 3.6.1–Section 3.6.4	1500
Channel implementation and proof rules	Section 3.6.5–Section 3.6.6	350
Tactics for symbolic execution	Section 3.7.2	500
Utilities (linked lists, permutations, *etc.*)	n.a.	450
Authoritative contribution ghost theory	Section 3.4	150
Recursive domain equation theory solver	Section 3.6.8	100
Examples:		
• Basic examples	Section 3.1 and Section 3.4.1	400
• Coarse-grained distributed merge sort	Section 3.2.3–Section 3.2.7	250
• Fine-grained distributed merge sort	Section 3.2.8	300
• Mapper with swapping	Section 3.3.2	400
• List reversal	Section 3.3.3	100
• Distributed mapper	Section 3.4.2	200
• Distributed map-reduce	Section 3.5	300
Total		5000

Figure 3.20: Overview of the Actris Coq mechanisation.

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

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

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

$$
\begin{aligned}
&\texttt{prog\_ref\_swap\_loop} := \lambda().\ \texttt{let}\ c := \texttt{start}\ (\texttt{rec}\ \mathit{go}\ c' := \texttt{let}\ \ell := \texttt{recv}\ c'\ \texttt{in}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \ell \leftarrow !\,\ell + 2;\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{send}\ c'\ ();\ \mathit{go}\ c')\ \texttt{in}\\
&\qquad\qquad \texttt{let}\ \ell_1 := \texttt{ref}\ 18\ \texttt{in}\\
&\qquad\qquad \texttt{let}\ \ell_2 := \texttt{ref}\ 20\ \texttt{in}\\
&\qquad\qquad \texttt{send}\ c\ \ell_1;\ \texttt{send}\ c\ \ell_2;\\
&\qquad\qquad \texttt{recv}\ c;\ \texttt{recv}\ c;\\
&\qquad\qquad !\,\ell_1 + !\,\ell_2
\end{aligned}
$$

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

Figure 3.21: Proof of message-passing program

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

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

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

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

The full Coq proof of the program is shown in Figure 3.21. The proved lemma is logically equivalent to the specification $\{\mathsf{True}\}\ \texttt{prog\_ref\_swap\_loop}\ ()\ \{v.\, v = 42\}$, but is presented in weakest precondition style as is common in Iris in Coq. We start the proof on line 4 by introducing the postcondition continuation `Φ`, and the hypothesis `HΦ: Φ #42`, and then evaluate the lambda expression with `wp_lam`. On line 5 we apply the specification `start_chan_spec` for `start _` by picking the expected protocol `prot_ref_loop`. This leaves us with two subgoals, separated by bullets `-`: one for the forked-off thread, and one for the main thread.

In the proof of the recursively-defined forked-off thread, we use `iLöb as "IH"` for Löb induction on line 6. This leaves us with the following intermediate proof state:

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

We now resolve the application of `c` to the recursive function with `wp_lam`. This lets us strip the later from the Löb induction hypothesis, as the program has taken a step. For brevity's sake we refer to the recursive program as `prog_rec`, in the following proof states.

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

On line 7 we resolve the proof of the body of the recursive function. So far, the proof only used Iris's standard tactics, we now use the Actris tactic for receive `wp_recv (l x) as "Hl"`, to resolve the receive in evaluation position, introducing the received logical variables `l` and `x`, along with the predicate of the protocol `l ↦ #x` naming it `Hl`. To do so, the protocol is normalised, unfolding the recursive definition once, as well as resolving the dualisation of the head, turning it into a receive as expected. This leads to the following proof state:

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

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

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

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

We finally close the proof of the forked-off thread on line 8. We first take two pure evaluation steps revolving the sequencing of operations with `do 2 wp_pure _` to reach the recursive call. This results in the proof state:

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

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

The proof of the main thread follows similarly. On line 9 we use `wp_alloc l1 as "Hl1"` and `wp_alloc l2 as "Hl2"`, to resolve the allocations of the new locations, binding the logical variables of the locations to `l1` and `l2`, and adding hypotheses `"Hl1"` and `"Hl2"` for ownership of these locations to the separation logic proof context. The proof state is then:

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

On line 10, we then resolve the first send operation with Actris's tactic `wp_send with "[$Hl1]"`, by giving up ownership of the location `l1`. Here, the protocol is normalised by unfolding the recursive definition, after which the head symbol is a send (**!**) as expected. The resulting proof state is as follows:

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

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

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

On line 11 we then proceed as expected with `wp_recv as "Hl1"` and `wp_recv as "Hl2"`, to resolve the receive operations, giving us back the updated point-to resources:

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

At line 12 we then continue by using `wp_load` twice to dereference the reacquired and updated locations, and then use trivial symbolic execution to resolve the remaining computations. On line 13 we finally close the proof with the continuation hypothesis `by iApply "HΦ".`

3.7.3 Tactic Support for Subprotocols

While the Actris tactics automatically apply the subprotocol rules during symbolic execution, as shown in Section 3.7.2, we sometimes want to prove subprotocol relations as explicit lemmas. We have tactic support for such proofs as well, which is integrated with the existing MoSeL tactics `iIntros`, `iExists`, `iFrame`, `iModIntro`, and `iSplitL/iSplitR` by automatically using the subprotocol rules to turn the goal into a goal where the regular Iris tactics apply.

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

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

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

Figure 3.22: Proof of subprotocol relation

To demonstrate these tactics, we will go through a proof of the subprotocol relation for the list reversing service presented in Section 3.3.3:

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

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

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

The full Coq proof is shown in Figure 3.22. On line 7 we start the proof by introducing the logical variables `l`, `xs` and the payload `llistI IT l xs` of the weaker protocol with the tactic `iIntros (l xs) "Hl"`. This tactic will implicitly apply the rule ⊑-SEND-IN, so the goal starts with a universal quantification `∀ (l : loc)(xs : list T). llistI IT l xs -∗...`, which is then introduced based on the regular Iris introduction pattern. This gives us:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.8 Related Work

As Actris combines results from both the separation logic and session types community, there is an abundance of related work. This section briefly elaborates on the relation to message passing in separation logic (Section 3.8.1) and process calculi (Section 3.8.2), session types (Section 3.8.3), endpoint sharing (Section 3.8.4), and verification efforts of map-reduce (Section 3.8.5).

3.8.1 Message Passing and Separation Logic

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

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

The original Iris [Jung et al. 2015] includes a small message-passing language with channels that do not preserve message order. It was included to demonstrate that

Iris is flexible enough to handle other concurrency models than standard shared-memory concurrency. Since the Hoare-triples for send and receive only reason about the entire channel buffer, protocol reasoning must be done via STSs or other forms of ghost state.

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

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

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

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

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

3.8.2 Separation Logic and Process Calculi

Another approach is to verify message-passing programs written in some dialect of process calculus. We focus on related work that combines process calculus with separation logic. Neither of the approaches below support delegation or concurrency paradigms other than message passing.

Francalanza et al. [2011] use separation logic to verify programs written in a CCS-like language. Channels model memory location, which has the effect that their

input-actions behave a lot like our updates of mutable state with variable substitutions updating the state. As a proof of concept they prove the correctness of an in-place quick-sort algorithm.

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

3.8.3 Session Types

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

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

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

Thiemann and Vasconcelos [2020] introduced label dependent session types, where tails of protocols can depend on the communicated message, which allows for encoding of the choice operators using send and receive. This is similar to our encoding of the choice operators in terms of Actris's dependent send and receive (Section 3.2.5).

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

Session subtyping for recursive type systems is universally carried out as a type simulation on infinite trees [Gay and Hole 2005; Mostrous et al. 2009; Mostrous and Yoshida 2015], which complicates subtyping under the recursive operator. Gay et al. [2020] provide further insights on this problem, although they investigate duality rather than subtyping. To reason about recursive types, Brandt and Henglein [1998] present a coinductive formulation of subtyping (which they apply to regular type systems, rather than session types). We use a similar coinductive formulation, but instead of ordinary coinduction, we use Iris's support for guarded recursion, which lets us prove subtyping relations of recursive protocols using Löb induction.

3.8.4 Endpoint Sharing

One of the key features of session types is that endpoints are owned by a single process. While these endpoints can be delegated (*i.e.,* transferred from one process to another), they typically cannot be shared (*i.e.,* be accessed by multiple processes concurrently). However, as we demonstrate in Section 3.4, sharing channels endpoints is often desirable, and possible in Actris.

In the pi calculus community there has been prior work on endpoint sharing, *e.g.,* by Atkey et al. [2016]; Kobayashi [2006]; Padovani [2014]. The latest contribution in this line of work is by Balzer and Pfenning [2017]; Balzer et al. [2019], who developed a type system based on session types with support for manifest sharing. Manifest sharing is the notion of sharing a channel endpoint between multiple processes using a lock-like structure to ensure mutual exclusion. Their key idea to ensure mutual exclusion using a type system is to use adjoint modalities to connect two classes of types: types that are linear, and thus denote unique channel ownership, and types that are unrestricted, and thus can be shared. The approach to endpoint sharing in Actris is different: dependent separation protocols do not include a built-in notion for endpoint sharing, but can be combined with Iris's general-purpose mechanisms for sharing, like locks.

3.8.5 Verification of Map-Reduce

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

3.9 Conclusion and Future Work

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

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

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

Chapter 4

Semantic Session Typing

Abstract Session types—a family of type systems for message-passing concurrency—have been subject to many extensions, where each extension comes with a separate proof of type safety. These extensions cannot be readily combined, and their proofs of type safety are generally not machine checked, making their correctness less trustworthy. We overcome these shortcomings with a semantic approach to binary asynchronous affine session types, by developing a logical relations model in Coq using the Iris program logic. We demonstrate the power of our approach by combining various forms of polymorphism and recursion, asynchronous subtyping, references, and locks/mutexes. As an additional benefit of the semantic approach, we demonstrate how to manually prove typing judgements of racy, but safe, programs that cannot be type checked using only the rules of the type system.

4.1 Introduction

Session types [Honda et al. 1998] guarantee that message-passing programs comply with a protocol (*session fidelity*), and do not crash (*type safety*). While session types are an active research area, we believe the following challenges have not received the attention that they deserve:

1. There are many extensions of session types with *e.g.,* polymorphism [Gay 2008], asynchronous subtyping [Mostrous et al. 2009], and sharing via locks [Balzer and Pfenning 2017]. While type safety has been proven for each extension in isolation, existing proofs cannot be readily composed with each other, nor with other substructural type systems like Affe, Alms, Linear Haskell, Plaid, Rust, Mezzo, Quill, or System F°.
2. Session types use substructural types to enforce a strict discipline of channel ownership. While conventional session-type systems can type check many functions, they inherently exclude some functions that do not obey the ownership discipline, even if they are safe.
3. Only few session-type systems and their safety proofs have been machine checked by a proof assistant, making their correctness less trustworthy.

We address these challenges by eschewing the traditional *syntactic approach* to type safety (using *progress and preservation*) and instead embrace the *semantic approach* to type safety [Appel and McAllester 2001; Ahmed 2004; Ahmed et al. 2010], using *logical relations* defined in terms of a program logic [Appel et al. 2007; Dreyer et al. 2009, 2019].

The semantic approach addresses the challenges above as (1) typing judgements are definitions in the program logic, and typing rules are lemmas in the program logic (they are not inductively defined), which means that extending the system with new typing rules boils down to proving the corresponding typing lemmas correct; (2) safe functions that cannot be conventionally type checked can still be semantically type checked by manually proving a typing lemma (3) all of our results have been mechanised in Coq using the Iris framework for concurrent separation logic [Jung et al. 2015, 2016; Krebbers et al. 2017a; Jung et al. 2018b; Krebbers et al. 2017b, 2018] giving us a high degree of trust that they are correct.

The syntactic approach to type safety requires global proofs of progress (well-typed programs are either values or can take a step) and preservation (steps taken by the program do not change types), culminating in type safety (well-typed programs do not get stuck). One key selling point of the semantic approach to type safety is that it does not require progress and preservation proofs, thereby allowing snippets of safe code to be type checked without requiring well-typed terms mid execution. Safety proofs are deferred to the program logic, whose adequacy/soundness theorem states that proving a program correct for any postcondition implies that the code will never get stuck. A concrete example of a racy program that can be semantically, but not conventionally, type checked is:

$$\lambda c.\, (\texttt{recv}\ c \mid\mid \texttt{recv}\ c)\ :\ \texttt{chan}\ (\texttt{?Z}.\, \texttt{?Z}.\, \texttt{end}) \multimap (\mathbf{Z} \times \mathbf{Z})$$

Two values are requested over channel c in parallel, and returned as a tuple (using the operator $\mid\mid$ for parallel composition, and the type $\texttt{chan}\ (\texttt{?Z}.\, \texttt{?Z}.\, \texttt{end})$ for a channel that expects to receive two integers). This program cannot be type checked using conventional session-type systems as channels are substructural types and cannot be owned by multiple threads at the same time. Nevertheless, this program is safe[1]—the order in which the values are received is irrelevant, as they have the same type.

The fact that this program cannot be type checked is not a shortcoming of conventional session-type systems. Since the correctness relies on a subtle argument (the `recv` is executed *exactly* twice in parallel), it is unreasonable to expect having syntactical typing rules that account for it. However, using the semantic approach, we can prove the corresponding typing lemma using the full power of the program logic.

An important prerequisite for proving typing lemmas such as the above is to use an expressive program logic. The Iris concurrent separation logic [Jung et al. 2015, 2016; Krebbers et al. 2017a; Jung et al. 2018b] has proved to be sufficiently expressive to define semantic type systems for *e.g.,* Rust [Jung et al. 2018a, 2021] and Scala [Giarrusso et al. 2020], due to its state-of-the-art built-in support for *e.g.,* resource ownership, recursion, polymorphism, and concurrency. In addition, we make

[1]For simplicity, we assume `recv` to be atomic, or a lock is needed. Even with a lock, conventional session-type systems cannot handle this program.

use of the Actris framework for message passing in Iris [Hinrichsen et al. 2020, 2021a]. Actris includes the notion of *dependent separation protocols*, which are like session types in structure, but were developed to prove functional correctness of message-passing programs. An additional advantage of Iris (and Actris) is that they come with an existing mechanisation in Coq. This mechanisation not only includes an adequacy/soundness theorem, but also tactical support for separation logic proofs [Krebbers et al. 2017b, 2018].

Contributions and Outline This paper presents an extensive machine-checked and semantic account of binary (two-party) asynchronous (sends are non-blocking) affine (resources may be discarded) session types. It makes the following contributions:

- We define a semantic session-type system as a logical relation in Iris using Actris's notion of dependent separation protocols (Section 4.2). As an additional conceptional contribution, this construction provides a concise connection between session types and separation logic.
- We demonstrate the extensibility of our approach by adding subtyping for term and session types, copyable types, equi-recursive term and session types, polymorphic term and session types, and mutexes (Section 4.3).
- We demonstrate the benefit of our type system being semantic by integrating the manual verification of safe but not conventionally type-checkable programs (Section 4.4).
- We provide insight on the benefits of a semantic type system in regards to mechanisation efforts (Section 4.5). All of our results are mechanised in the Coq proof assistant and can be found in [Hinrichsen et al. 2021b].

4.2 A Tour of Semantic Session Typing

We show how to build a semantic session-type system using logical relations on top of an untyped concurrent language with message passing (Section 4.2.1). We provide a brief overview of Iris (Section 4.2.2), and then present a lightweight affine type system (Section 4.2.3) as the core upon which we built our session-type system (Section 4.2.4). Our affine type system is inspired by RustBelt [Jung et al. 2018a, 2021], but drops Rust-specific features like borrowing and lifetimes to focus on session types.

4.2.1 Language

We use an untyped higher-order functional programming language with concurrency, mutable references, and binary asynchronous message passing, whose syntax is:

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

We let $b \in \mathbb{B}$, $i \in \mathbb{Z}$, $\ell \in \mathsf{Loc}$, and $c \in \mathsf{Chan}$, where Loc and Chan are countably infinite sets of identifiers. We omit the standard operations on pairs, sums, *etc.* We write $\lambda x.\, e$ for $\texttt{rec}\ \_\ x := e$, and $\texttt{let}\, x := e_1\, \texttt{in}\, e_2$ for $(\lambda x.\, e_2)\ e_1$, and $e_1; e_2$ for $\texttt{let}\ \_ := e_1\, \texttt{in}\, e_2$. Message passing is given an asynchronous semantics: $\texttt{new\_chan}\ ()$ returns a pair (c_1, c_2) of channel endpoints that operate on buffers $(\vec{v}_1, \vec{v}_2)$ that are initially empty, $\texttt{send}\ c_i\ w$ enqueues message w in $\vec{v}_i$, while $\texttt{recv}\ c_i$ blocks until a message w is available in $\vec{v}_{(\text{if } i=2 \text{ then } 1 \text{ else } 2)}$, and then dequeues and returns w. Mutable references ℓ are allocated with $\texttt{ref}\ e$, updated using $e_1 \leftarrow e_2$, and dereferenced with $!\, e$. Parallel composition $e_1 \mid\mid e_2$ executes e_1 and e_2 in parallel and returns the results as a tuple, once they have terminated. The language also supports fork and compare-and-set.

4.2.2 Semantic Typing in Iris

The idea of semantic typing is to represent types as *logical relations*, which are predicates that describe the values that inhabit the type. To model type systems with features like references or session types, these predicates need to range over program states. To avoid threading through program states explicitly, we do not use ordinary set-theoretic predicates, but use predicates in a program logic, and use the connectives of the program logic to give concise definitions of types. The program logic that we use is Iris, whose propositions $P, Q \in \mathsf{iProp}$ implicitly range over an extensible notion of resources, which includes the program state.

Iris is a higher-order separation logic, so it has the usual logical connectives such as conjunction ($P \wedge Q$), implication ($P \Rightarrow Q$), universal ($\forall x : \tau.\, P$) and existential ($\exists x : \tau.\, P$) quantification, as well as the connectives of separation logic:

- The *points-to connective* ($\ell \mapsto v$) asserts exclusive resource ownership of a heap location $\ell \in \mathsf{Loc}$, stating that it holds the value $v \in \mathsf{Val}$.
- The *separating conjunction* ($P * Q$) states that P and Q holds for disjoint sets of resources.
- The *separating implication* ($P \mathrel{-\!\!*} Q$) states that by giving up ownership of the resources described by P, we obtain ownership of the resources described by Q. Separating implication is used similarly to implication since (P entails $Q \mathrel{-\!\!*} R$) iff ($P * Q$ entails R).
- The *weakest precondition* ($\mathsf{wp}\ e\ \{\Phi\}$) states that given a postcondition $\Phi : \mathsf{Val} \to \mathsf{iProp}$, the expression e is safe, and, if e reduces to a value v, then $\Phi\, v$ holds. We write $\mathsf{wp}\ e\ \{w.\, Q\}$ for $\mathsf{wp}\ e\ \{\lambda w.\, Q\}$.

As we see in Section 4.2.3 these connectives match up with the type formers for unique references ($\ell \mapsto v$), products ($P * Q$), and affine functions ($P \mathrel{-\!\!*} Q$ and $\mathsf{wp}\ e\ \{\Phi\}$).

Iris's notion of resources is not limited to heap locations, but can be extended with custom resources. This feature is used by Actris to extend Iris with support for reasoning about functional correctness of message-passing programs (Section 4.2.4) by means of the connective ($c \rightarrowtail -$) that asserts exclusive resource ownership of the channel c. Moreover, Iris has an extensible mechanism of *ghost* resources, which we use in this paper to semantically type safe yet not conventionally type-checkable programs (Section 4.4).

To define recursive types semantically (Section 4.3.3), Iris provides the *later modality* ($\triangleright P$) and the *guarded fixpoint operator* ($\mu x : \tau.\, t$), which enable guarded recursive definitions of Iris propositions and terms. The guarded fixpoint operator requires all recursive occurrences of the variable x to occur *guarded* in t, where an occurrence is guarded if it appears below a $\triangleright$ modality. This ensures that t is *contractive* in the variable x, which guarantees that a unique fixpoint exists. Guarded fixpoints can be folded and unfolded using the equality $\mu(x : \tau).\, t = t[(\mu(x : \tau).\, t)/x]$.

The proposition $\triangleright P$ is strictly weaker than P, since P entails $\triangleright P$, while the reverse does not hold. The $\triangleright$ modality can be eliminated by taking a program step, which is formalised by the Iris proof rule: $(\triangleright P) * \mathsf{wp}\ e\ \{\Phi\} \mathrel{-\!\!*} \mathsf{wp}\ e\ \{w.\, P * \Phi\, w\}$ if $e \notin \mathsf{Val}$ and $w \notin FV(P)$. This rule indicates that $\triangleright P$ can also be read as "P holds after one more step of computation", as P is obtained without $\triangleright$ modality in the postcondition of the weakest precondition, denoting that at least one step has been taken.

In this paper we will not further detail the semantics of Iris, but refer the interested reader to Jung et al. [2018b] for an extensive account of the Iris model and proof rules.

4.2.3 Term Types

The definitions of our semantic type system are shown in Figure 4.1. Types Type_k are indexed by kinds; ★ for *term types*, and ⧫ for *session types*. Meta-variables $A, B \in \mathsf{Type}_\bigstar$ are used for term types, $S \in \mathsf{Type}_\blacklozenge$ for session types, and $K \in \mathsf{Type}_k$ for types of any kind. Term types $\mathsf{Type}_\bigstar$ are defined as Iris predicates over values, and session types $\mathsf{Type}_\blacklozenge$ are defined as dependent separation protocols of Actris (Section 4.2.4).

Type Formers The ground types (unit type $\mathbf{1}$, Boolean type $\mathbf{B}$, and integer type $\mathbf{Z}$) are defined through membership of the corresponding set ($\{()\}$, $\mathbb{B}$, and $\mathbb{Z}$). The type former $\mathtt{ref}_{\mathtt{uniq}}\, A$ for uniquely-owned references, $A_1 \times A_2$ for products, and $A \multimap B$ for affine functions nicely demonstrate the advantage of separation logic—since types are Iris predicates, they implicitly describe which resources are owned. The points-to connective $(w \mapsto v)$ is used to describe that $\mathtt{ref}_{\mathtt{uniq}}\, A$ consists of the locations $w \in \mathsf{Loc}$ that hold a value $v \in \mathsf{Val}$ for which the resources $A\, v$ are owned. The separating conjunction $(*)$ is used to describe that $A_1 \times A_2$ consists of tuples (w_1, w_2), where the resources $A_1\, w_1$ and $A_2\, w_2$ are owned *separately*. The separating implication $(-\!\!*)$ and weakest precondition are used to describe that the affine function type $A \multimap B$ consists of values w that when applied to an argument v consume the resources $A\, v$, and in return, produce the resources B for the result of $w\, v$. Note that the weakest precondition $\mathsf{wp}\ (w\ v)\ \{B\}$ is used so we can talk about the result of $w\, v$. We could not use $B\ (w\, v)$ since the term $w\, v$ is not a value.

We use Iris's later modality ($\triangleright$) to ensure that type formers are contractive, which is needed to model equi-recursive types using Iris's guarded fixpoint operator in Section 4.3.3.

Typing Judgement As is common in substructural type systems with operations that perform strong updates, we use a typing judgement $\Gamma \vDash e : A \dashv \Gamma'$ (defined in Figure 4.1) with a *pre-* and *post-context* $\Gamma, \Gamma' \in \mathsf{List}(\mathsf{String} \times \mathsf{Type}_\bigstar)$. These contexts describe the types of variables before and after execution of the expression e.

Term types:

$$
\begin{aligned}
\mathsf{Type}_\bigstar &\triangleq \mathsf{Val} \to \mathsf{iProp} \\
\mathtt{any} &\triangleq \lambda w.\, \mathsf{True} \\
\mathbf{1} &\triangleq \lambda w.\, w \in \{()\} \\
\mathbf{B} &\triangleq \lambda w.\, w \in \mathbb{B} \\
\mathbf{Z} &\triangleq \lambda w.\, w \in \mathbb{Z} \\
\mathtt{ref}_{\mathtt{uniq}}\, A &\triangleq \lambda w.\, \exists v.\, w \in \mathsf{Loc} * (w \mapsto v) * \triangleright (A\, v) \\
A_1 \times A_2 &\triangleq \lambda w.\, \exists v_1, v_2.\, w = (v_1, v_2) \;\; * \triangleright (A_1\, v_1) * \triangleright (A_2\, v_2) \\
A_1 + A_2 &\triangleq \lambda w.\, \exists v.\, (w = \mathtt{inl}\ v * \triangleright (A_1\, v)) \lor (w = \mathtt{inr}\ v * \triangleright (A_2\, v)) \\
A \multimap B &\triangleq \lambda w.\, \forall v.\, \triangleright (A\, v) \mathrel{-\!\!*} \mathsf{wp}\ (w\ v)\ \{B\} \\
\mathtt{chan}\ S &\triangleq \lambda w.\, w \rightarrowtail S
\end{aligned}
$$

Session Types:

$$
\begin{aligned}
\mathsf{Type}_\blacklozenge &\triangleq \mathsf{iProto} \\
\mathtt{end} &\triangleq \mathbf{end} \\
\mathbf{!}A.\, S &\triangleq \mathbf{!}\, (v : \mathsf{Val})\ \langle v \rangle \{A\, v\}.\, S \\
\mathbf{?}A.\, S &\triangleq \mathbf{?}(v : \mathsf{Val})\ \langle v \rangle \{A\, v\}.\, S \\
\oplus\{\vec{S}\} &\triangleq \mathbf{!}\, (l : \mathbb{Z})\ \langle l \rangle \left\{ l \in \mathsf{dom}(\vec{S}) \right\}.\, \vec{S}(l) \\
\&\{\vec{S}\} &\triangleq \mathbf{?}(l : \mathbb{Z})\ \langle l \rangle \left\{ l \in \mathsf{dom}(\vec{S}) \right\}.\, \vec{S}(l)
\end{aligned}
$$

Judgements:

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

Figure 4.1: Typing judgements and type formers of the semantic type system.

As is standard in logical relations, we use *closing substitutions* to give a semantics to typing contexts. Closing substitutions $\sigma \in \mathsf{String} \xrightarrow{\mathsf{fin}} \mathsf{Val}$ are finite partial functions that map the free variables of an expression to corresponding values. Closing substitutions come with a judgement $\Gamma \vDash \sigma$, which expresses that the closing substitution σ is well-typed in the context Γ. The definition of this judgement employs the iterated separation conjunction $\mathop{\Large *}_{(x,A)\in\Gamma}$ to ensure that for each variable typing (x, A) in Γ, there is a corresponding value in the closing substitution $\sigma(x)$ for which the resources $A(\sigma(x))$ are owned separately.

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

Selection of Iris's proof rules for weakest preconditions:

$$\Phi\, v \twoheadrightarrow \mathsf{wp}\, v\, \{\Phi\} \qquad \text{(WP-VAL)}$$

$$\ell \mapsto v \twoheadrightarrow \mathsf{wp}\, !\,\ell\, \{w.\, (v = w) * (\ell \mapsto v)\} \qquad \text{(WP-LOAD)}$$

$$\mathsf{wp}\, e_1\, \{v.\, \mathsf{wp}\, e_2[v/x]\, \{\Phi\}\} \twoheadrightarrow \mathsf{wp}\, (\texttt{let}\, x := e_1\, \texttt{in}\, e_2)\, \{\Phi\} \qquad \text{(WP-LET)}$$

$$\mathsf{wp}\, e_1\, \{\Phi_1\} * \mathsf{wp}\, e_2\, \{\Phi_2\} \twoheadrightarrow \mathsf{wp}\, (e_1 \mid\mid e_2)\, \left\{ w.\, \exists w_1, w_2.\, \begin{array}{l} (w = (w_1, w_2)) * \\ \Phi_1\, w_1 * \Phi_2\, w_2 \end{array} \right\} \qquad \text{(WP-PAR)}$$

Selection of semantic typing rules:

$$\Gamma \vDash i : \mathbf{Z} \dashv \Gamma \qquad\qquad \Gamma, x : A \vDash x : A \dashv \Gamma, x : \mathtt{any}$$

$$\Gamma, x : \mathtt{ref}_{\mathtt{uniq}}\, A \vDash\, !\, x : A \dashv \Gamma, x : \mathtt{ref}_{\mathtt{uniq}}\, \mathtt{any}$$

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

$$\frac{\Gamma_1 \vDash e_1 : A_1 \dashv \Gamma_1' \qquad \Gamma_2 \vDash e_2 : A_2 \dashv \Gamma_2'}{\Gamma_1 \cdot \Gamma_2 \vDash e_1 \mid\mid e_2 : A_1 \times A_2 \dashv \Gamma_1' \cdot \Gamma_2'}$$

Figure 4.2: A selection of Iris's proof rules and semantic typing rules.

Typing Rules Now that the type formers and the typing judgement are in place, we state the conventional typing rules as lemmas. We prove these lemmas by unfolding the definition of the semantic typing judgement $\Gamma \vDash e : A \dashv \Gamma'$, and proving the corresponding proposition in Iris using the rules for weakest preconditions. A selection of typing rules, along with Iris's weakest precondition rules used to prove them, is presented in Figure 4.2.

The typing rule for integer literals follows immediately from WP-VAL, which states that the weakest precondition of a value v holds if the postcondition $\Phi\, v$ holds. The typing rule for variables also uses WP-VAL. Since the pre-context is $\Gamma, x : A$, we can assume ownership of $A\, v$ for some value v, and should prove a weakest precondition for v. After using WP-VAL, we prove the postcondition by giving up $A\, v$. Note that the post-context is $\Gamma, x : \mathtt{any}$ as ownership of A has been moved out. For substructural type systems this is crucial as in expressions such as $\texttt{let}\, x := y\, \texttt{in}\, e$, it is generally not allowed to use y in e as ownership of the type of y has moved to x. This is formalised by giving the variable y type $\mathtt{any}$ in e. The typing rules for load, let, and parallel composition are proved using the Iris rules WP-LOAD, WP-LET, and WP-PAR. The rule for parallel composition moreover relies on the property $(\Gamma_1 \cdot \Gamma_2 \vDash \sigma)$ iff $(\Gamma_1 \vDash \sigma) * (\Gamma_2 \vDash \sigma)$, which allows us to subdivide and recombine ownership of the pre- and post-contexts between both operands.

Type Safety Type safety means: if $[\,] \vDash e : A \dashv \Gamma$, then e is safe, *i.e.,* e will not get stuck w.r.t. the language's operational semantics. For syntactic type systems, type safety is usually proven via the progress and preservation theorems. For our

semantic type system, we get type safety from Iris's adequacy theorem, which states that a closed proof of a weakest precondition implies safety [Krebbers et al. 2017a; Jung et al. 2018b]. Note that our type system is affine (resources are not explicitly deallocated), and thus the post-context Γ in the type safety statement need not be empty. We use an affine type system as that allows more practical safe programs to be typeable.

4.2.4 Session Types

We extend our core type system with the basic session-type formers for sending a message $!A.\, S$, receiving a message $?A.\, S$, the choice primitives for selection $\oplus\{\vec{S}\}$ and branching $\&\{\vec{S}\}$, and the terminator `end`. We let $\vec{S} : \mathbb{Z} \xrightarrow{\text{fin.}} \mathsf{Type}_\blacklozenge$ be finite partial functions from labels to session types, and often write $\vec{S} = l_1 : S_1, \ldots l_n : S_n$. The term type `chan` S dictates that a term is a channel that follows the session type S.

Session types are defined in terms of Actris's *dependent separation protocols* [Hinrichsen et al. 2020], which are similar to session types in structure, but can express functional properties of the transferred data. Dependent separation protocols $prot \in \mathsf{iProto}$ are streams of $!\,\vec{x} : \vec{\tau}\,\langle v\rangle\{P\}.\, prot$ and $?\vec{x} : \vec{\tau}\,\langle v\rangle\{P\}.\, prot$ constructors that are either infinite or finite. Here, v is the value that is being sent or received, P is an Iris proposition denoting the ownership of the resources being transferred as part of the message, and the logical variables $\vec{x} : \vec{\tau}$ bind into v, P, and $prot$ to constrain the message v and the tail protocol $prot$. Finite protocols are ultimately terminated by an **end** constructor. As an example, the dependent separation protocol $!\,(\ell : \mathsf{Loc})\,(i : \mathbb{Z})\,\langle \ell\rangle\{\ell \mapsto i * 10 \leq i\}.\, ?\langle()\rangle\{\ell \mapsto (i+1)\}.\, \mathbf{end}$ expresses that an integer reference whose value is at least 10 is sent, after which the recipient increments it by one and sends back a unit token () along with the reference ownership.

Actris's connective $c \rightarrowtail prot$ denotes ownership of a channel c with a dependent separation protocol $prot$. The Actris proof rules are shown in Figure 4.3. The rule for `new_chan` () allows ascribing any protocol to a new channel, obtaining ownership of $c \rightarrowtail prot$ and $c' \rightarrowtail \overline{prot}$ for the respective endpoints. Here, $\overline{prot}$ is the *dual* of $prot$, in which any receive (**?**) is turned into a send (**!**), and *vice versa*. The rule for `send` c w requires the head of the protocol to be a send (**!**), and the value w that is sent to match up with the ascribed value. Concretely, to send a message w, one needs to give up ownership of $c \rightarrowtail !\,\vec{x} : \vec{\tau}\,\langle v\rangle\{P\}.\, prot$, pick an appropriate instantiation $\vec{t}$ for the variables $\vec{x} : \vec{\tau}$ so that $w = v[\vec{t}/\vec{x}]$, and give up ownership of the associated resources $P[\vec{t}/\vec{x}]$. Subsequently, one gets back ownership of the protocol tail $c \rightarrowtail prot[\vec{t}/\vec{x}]$. The rule for `recv` c is essentially dual to the rule for `send` c w. One needs to give up ownership of $c \rightarrowtail ?\vec{x} : \vec{\tau}\,\langle v\rangle\{P\}.\, prot$, and in return 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 $prot[\vec{y}/\vec{x}]$, where $\vec{y}$ are instances of the variables of the protocol.

Semantics of Session Types The definitions of session types are shown in Figure 4.1. Since session types ($\mathsf{Type}_\blacklozenge$) are defined as dependent separation protocols iProto, the channel type `chan` S is defined in terms of Actris's connective for channel ownership $w \rightarrowtail S$. The definition of the terminator (`end`), send (**!**), and receive (**?**)

Actris's proof rules for dependent separation protocols:

$$\mathsf{wp}\,\texttt{new\_chan}\ ()\ \{w.\,\exists c, c'.\,(w = (c, c')) * c \rightarrowtail \mathit{prot} * c' \rightarrowtail \overline{\mathit{prot}}\}$$

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

$$c \rightarrowtail \mathbf{?}\,\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot} \mathrel{-\!\!*} \mathsf{wp}\,\texttt{recv}\ c\{w.\,\exists \vec{y}.\,(w = v[\vec{y}/\vec{x}]) * c \rightarrowtail \mathit{prot}[\vec{y}/\vec{x}] * P[\vec{y}/\vec{x}]\}$$

Semantic typing rules for channels:

$$\Gamma \vDash \texttt{new\_chan}\ () : \mathtt{chan}\ S \times \mathtt{chan}\ \overline{S} \dashv \Gamma$$

$$\frac{\Gamma \vDash e : A \dashv \Gamma', x : \mathtt{chan}\ (\mathbf{!}A.\,S)}{\Gamma \vDash \texttt{send}\ x\ e : \mathbf{1} \dashv \Gamma', x : \mathtt{chan}\ S} \qquad \Gamma, x : \mathtt{chan}\ (\mathbf{?}A.\,S) \vDash \texttt{recv}\ x : A \dashv \Gamma, x : \mathtt{chan}\ S$$

$$\frac{1 \le i \le n}{\Gamma, x : \mathtt{chan}\ (\oplus\{l_1 : S_1, \ldots, l_n : S_n\}) \vDash \texttt{select}\ x\ l_i : \mathbf{1} \dashv \Gamma, x : \mathtt{chan}\ S_i}$$

$$\frac{\Gamma, x : \mathtt{chan}\ S_1 \vDash e_1 : A \dashv \Gamma' \qquad \cdots \qquad \Gamma, x : \mathtt{chan}\ S_n \vDash e_n : A \dashv \Gamma'}{\Gamma, x : \mathtt{chan}\ (\&\{l_1 : S_1, \ldots, l_n : S_n\}) \vDash \texttt{branch}\ x\ \texttt{with}\ l_1 \Rightarrow e_1 \mid \ldots \mid l_n \Rightarrow e_n : A \dashv \Gamma'}$$

Figure 4.3: Actris's proof rules for dependent separation protocols and semantic typing rules for channels.

follow from their dependent separation protocol counterparts. For example $\mathbf{!}A.\,S$ is defined as $\mathbf{!}\,(v : \mathsf{Val})\,\langle v\rangle\{A\ v\}.\,S$. It says that a value v is sent along with ownership of the type predicate $A\ v$.

While the choice types $\oplus\{\vec{S}\}$ and $\&\{\vec{S}\}$ do not have a direct counterpart in Actris, they can be encoded. For example, $\oplus\{\vec{S}\}$ is defined as $\mathbf{!}\,(l : \mathbb{Z})\,\langle l\rangle\left\{l \in \mathsf{dom}(\vec{S})\right\}.\,\vec{S}(l)$. It expresses that a valid label $l \in \mathsf{dom}(\vec{S})$ (modelled as an integer) is sent. This definition makes use of the fact that dependent separation protocols are *dependent*, as the protocol tail $\vec{S}(l)$ depends on the label l that is sent.

Duality The duality $\overline{S}$ of session types S is inherited from Actris. We thus obtain the usual duality laws (on the left) from the Actris duality laws (on the right):

$$\overline{\mathtt{end}} = \mathtt{end} \qquad \overline{\mathbf{end}} = \mathbf{end}$$

$$\overline{\mathbf{!}A.\,S} = \mathbf{?}A.\,\overline{S} \qquad \overline{\mathbf{!}\,\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot}} = \mathbf{?}\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\,\overline{\mathit{prot}}$$

$$\overline{\mathbf{?}A.\,S} = \mathbf{!}A.\,\overline{S} \qquad \overline{\mathbf{?}\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot}} = \mathbf{!}\,\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\,\overline{\mathit{prot}}$$

Similarly, our semantic definition of the branch (&) and select ($\oplus$) operators in terms of Actris's send (**!**) and receive (**?**) protocols, enables us to use the Actris duality laws

to prove that the dual of a select is a branch, and *vice versa*:

$$\overline{\oplus\{l_1 : S_1, \ldots, l_n : S_n\}} = \&\{l_1 : \overline{S_1}, \ldots, l_n : \overline{S_n}\}$$
$$\overline{\&\{l_1 : S_1, \ldots, l_n : S_n\}} = \oplus\{l_1 : \overline{S_1}, \ldots, l_n : \overline{S_n}\}$$

Session Typing Rules The session typing rules are shown in Figure 4.3. Since the channel operations perform strong updates, the typing rules require channels to be variables so they can update the context. Given the close similarity between Actris and session typing, the typing rules follow from the Actris rules up to minor separation logic reasoning.

The rules for `select` and `branch` demonstrate the extensibility of our approach. Our language does not have these operations as primitives, but they can be defined as macros:

$$\texttt{select}\ x\ l \triangleq \texttt{send}\ x\ l$$
$$\begin{aligned}\texttt{branch}\ x\ \texttt{with}\ &\triangleq \texttt{let}\ i := \texttt{recv}\ x\ \texttt{in}\\ l_1 \Rightarrow e_1 \mid \ldots \mid\ &\quad\ \texttt{if}\ i = l_1\ \texttt{then}\ e_1\ \texttt{else}\ \cdots\\ l_n \Rightarrow e_n\ &\quad\ \texttt{if}\ i = l_n\ \texttt{then}\ e_n\ \texttt{else}\ ()\,()\end{aligned}$$

The typing rule for `select` follows immediately from the proof rule for `send`. Similarly, the typing rule for `branch` follows from the proof rule for `recv`, but additionally requires some reasoning in Iris about the sequence of if-expressions. Note that the stuck expression $()\,()$ is used in case no matching branch for the label l_i has been found. While this stuck expression is obviously not safe, it is never executed because of the condition $l \in \mathsf{dom}(\vec{S})$ in the definition of select ($\oplus$) and branch ($\&$).

Type Safety Since the extension with session types did not change the definition of the semantic typing judgement, but merely added new type formers and typing rules, the type safety result from Section 4.2.3 remains applicable without change.

4.3 Extending the Type System

We demonstrate the extensibility of our approach to session types by adding term- and session-level subtyping (Section 4.3.1 and Section 4.3.7), copyable types (Section 4.3.2), term- and session-level equi-recursive types (Section 4.3.3), term- and session-level polymorphism (Section 4.3.4 and Section 4.3.5), and locks/mutexes (Section 4.3.6). While we only present a small representative selection of rules associated with each extension, all rules can be found in Section 4.A.

4.3.1 Term-Level Subtyping

Subtyping $A <: B$ indicates that any member of type A is also a member of type B. In a semantic type system, subtyping is defined in terms of the separating implication:

$$A <: B \triangleq \forall v.\, A\ v \mathrel{-\!\!*} B\ v$$
$$\Gamma <:_{\mathbf{ctx}} \Gamma' \triangleq \forall \sigma.\, (\Gamma \vDash \sigma) \mathrel{-\!\!*} (\Gamma' \vDash \sigma)$$

The definition states that A is a subtype of B if for any value v, we can give up resources $A\ v$ to obtain resources $B\ v$. The *context subtyping relation* $\Gamma <:_{\mathbf{ctx}} \Gamma'$ is defined similarly. It is essentially the pointwise lifting of the subtyping relation, applied to each type in the contexts Γ and Γ'. It expresses that when we hold resources $\Gamma \vDash \sigma$ for the context Γ, then we can give those up to obtain the resources $\Gamma' \vDash \sigma$ for Γ'. With these definitions at hand, we prove the usual subsumption rule as a lemma:

$$\frac{\Gamma_1 <:_{\mathbf{ctx}} \Gamma_1' \qquad \Gamma_1' \vDash e : A \dashv \Gamma_2' \qquad A <: B \qquad \Gamma_2' <:_{\mathbf{ctx}} \Gamma_2}{\Gamma_1 \vDash e : B \dashv \Gamma_2}$$

The proof of the above lemma makes use of the Iris proof rule $(\forall v.\, \Phi_1\, v \mathrel{-\!\!*} \Phi_2\, v) \mathrel{-\!\!*} \mathsf{wp}\, e\, \{\Phi_1\} \mathrel{-\!\!*} \mathsf{wp}\, e\, \{\Phi_2\}$, which states that separating implications can be applied in the postconditions of weakest preconditions.

In addition to the subsumption rule, we prove the conventional subtyping rules as lemmas. For example:

$$A <: \mathtt{any} \qquad \frac{A <: B \qquad B <: C}{A <: C} \qquad \frac{C <: A \qquad B <: D}{A \multimap B <: C \multimap D} \qquad \frac{A <: C \qquad B <: D}{A \times B <: C \times D}$$

These lemmas are proved by unfolding the definition of the subtyping relation, and involve some trivial reasoning using separating implication in Iris. We will see more interesting subtyping rules in section 4.3.2 and section 4.3.7.

4.3.2 Copyable Types

Session-type systems are substructural, in the sense that some types are inhabited by values that can be used at most once. This becomes evident in the variable and load rules from section 4.2.3, which move out ownership by turning the element type into the `any` type. While moving out ownership is necessary for soundness in general, this is too restrictive for types that do not assert ownership of any resources, such as $\mathbf{B}$, $\mathbf{Z}$, or $\mathbf{Z} * \mathbf{B}$. These types need not be moved out as their inhabitants can be used multiple times. We therefore extend the type system with a notion of *copyable types*. Concretely, we define a type former `copy` and a property `copyable`:

$$\mathtt{copy}\, A \triangleq \lambda w.\ \square(A\, w)$$
$$\mathtt{copyable}\, A \triangleq A <: \mathtt{copy}\, A$$

The type $\mathtt{copy}\, A$ describes the values of type A that can be freely duplicated (used an arbitrary number of times). We thus have $A <: \mathtt{copy}\, A$ for ground types $A \in \{\mathbf{1}, \mathbf{B}, \mathbf{Z}\}$, but not for types like $A \in \{\mathtt{ref}_{\mathtt{uniq}}\, B, \mathtt{chan}\ S\}$ that assert ownership. Conversely, we have $\mathtt{copy}\, A <: A$ for any type A, *i.e.,* $\mathtt{copy}\, A$ is always a subtype of A. A type is *copyable* (written $\mathtt{copyable}\, A$) if *all* of its values can be freely duplicated, *i.e.,* when A is a subtype of $\mathtt{copy}\, A$. Ground types ($\mathbf{1}$, $\mathbf{B}$, $\mathbf{Z}$) are copyable, and copyability is closed under products and sums.

An example of a type where some, but not all, values can be duplicated is the type $A \multimap B$ of affine functions: a function can only be duplicated if it has not captured ownership of exclusive resources from the context (through a free variable that has

a non-copyable type). Hence, we define $A \to B \triangleq \texttt{copy}\,(A \multimap B)$ as the type of *unrestricted* functions, that can be applied any number of times.

The type former `copy` is defined using the *persistence* modality ($\Box$) of Iris, where $\Box\, P$ means that the proposition P holds without ownership of (exclusively-owned) resources. Propositions that do not assert ownership of (exclusively-owned) resources are called *persistent*. In particular, $\Box\, P$ is always persistent, allowing the proposition P to be freely duplicated using the rule $\Box\, P \twoheadrightarrow (\Box\, P * P)$. This allows copyable types occurring in the context to be duplicated:

$$x : A <:_{\mathbf{ctx}} x : A, x : A \qquad \text{if } \texttt{copyable}\, A$$

Our approach of using Iris's notion of persistence to model copyability of types is similar to the approach used in RustBelt [Jung et al. 2018a, 2021] to model the substructural features of Rust. However, copyability in RustBelt is defined directly in Iris, and not reflected into the type system by means of a `copy` type former and a subtyping rule.

4.3.3 Equi-Recursive Term and Session Types

We extend our type system with equi-recursive types using Iris's fixpoint operator. Recall from Section 4.2.2 that Iris's fixpoint operator requires that recursive definitions are contractive, meaning that recursive occurrences appear below a later modality ($\triangleright$). A recursive occurrence is also considered guarded when it appears in:

- The postcondition Φ of an Iris weakest precondition $\mathsf{wp}\, e\, \{\Phi\}$ with $e \notin \mathsf{Val}$.
- The tail *prot* of the dependent separation protocols $!\, \vec{x} : \vec{\tau}\, \langle v \rangle \{P\}.\, prot$ and $?\vec{x} : \vec{\tau}\, \langle v \rangle \{P\}.\, prot$.
- The protocol *prot* of the Actris connective $c \rightarrowtail prot$ for channel ownership.

These occurences are guarded because the corresponding constructs contain $\triangleright$ modalities internally.

We lift the guarded recursion operator of Iris into a *kinded* operator for equi-recursion in the type system:

$$\mu\, (X : k).\, K \triangleq \mu (X : \mathsf{Type}_k).\, K \qquad (K \text{ is contractive in } X)$$

From Iris's proof rule for fixpoints, we get that this is indeed a fixpoint, *i.e.,* we have $\mu\, (X : k).\, K = K[\mu\, (X : k).\, K / X]$.

We put later modalities in the definitions of type formers to ensure that they are contractive in all arguments. This allows construction of recursive term and session types, including examples from the session type literature [Gay et al. 2020], such as $\mu\, (X : \blacklozenge).\, !(\texttt{chan}\ X).\, X$, where the recursion variable X occurs in the type of messages.

It is worth noting that most existing logical relation developments in Iris model iso-recursive types. Hence, instead of putting $\triangleright$ modalities in the definitions of type formers, they put a $\triangleright$ modality in the definition of the recursion operator. This avoids the contractive side-condition, but requires explicit fold and unfold operations in the language (to take an operational step to remove the $\triangleright$ modality).

4.3.4 Polymorphism in Term Types

We extend the type system with kinded parametric polymorphism, by introducing universal types $\forall(X : k).\, A$ and existential types $\exists(X : k).\, A$, which are polymorphic in a variable X of kind k. The kind k indicates whether the type is polymorphic over term types (kind $\bigstar$) or session types (kind $\blacklozenge$). Using polymorphism in term types, we can write types such as $\forall(X : \bigstar).\, X \to X$ (for describing the polymorphic identity function). Using polymorphism in session types, we can write types such as $\forall(X : \blacklozenge).\, \texttt{chan}\ (\mathbf{!Z}.X) \multimap \texttt{chan}\ X$ (for describing a function that reads an integer from a channel with an arbitrary tail X). Universal and existential types are defined as follows:

$$\forall(X : k).\, A \triangleq \lambda w.\, \forall(X : \mathsf{Type}_k).\, \mathsf{wp}\ (w\ ())\ \{A\}$$
$$\exists(X : k).\, A \triangleq \lambda w.\, \exists(X : \mathsf{Type}_k).\, \triangleright(A\, w)$$

As is custom for logical relations in Iris, these types are defined in the style of parametricity—they use Iris-level universal and existential quantifiers over semantic types $X : \mathsf{Type}_k$. This is possible because Iris supports higher-order impredicative quantification (*i.e.,* quantification over Iris predicates and Actris protocols).

Note that universal types are inhabited by values w that produce a value of the instantiated type A when applied to the unit value $()$, as indicated by the weakest precondition in the definition. In other words, the inhabitants of universal types are *thunks*. This is since we consider a type system with explicit type abstraction and type instantiation constructs. Since the base language is untyped, we use term-level abstractions to indicate type abstraction and instantiation: type abstraction is written $\lambda\_.\, e$ and type instantiation is written $w\,()$ when w is a type abstraction. By using explicit thunks, we avoid having to impose an ML-like *value restriction* [Wright 1995] to ensure type safety in the presence of imperative side-effects. The typing rules for term-level polymorphism are standard and can be found in Section 4.A.

4.3.5 Polymorphism in Session Types

A more interesting extension is polymorphism in session types [Gay 2008]. An example is the following type, which describes the interaction with a polymorphic computation service:

$$\texttt{compute\_type} \triangleq \mu\,(\mathit{rec} : \blacklozenge).\ \oplus\{\texttt{cont} : \mathbf{!}_{(X:\bigstar)}\,(\mathbf{1} \multimap X).\, \mathbf{?}X.\, \mathit{rec},\ \texttt{stop} : \texttt{end}\}$$

The service can be used by sending computation requests $\mathbf{1} \multimap X$, and then awaiting their results X. Different types can be picked for the type variable X at each recursive iteration.

To extend our type system with polymorphism in session types, we redefine the send and receive session types to include binders $\vec{X}$ for type variables:

$$\mathbf{!}_{\vec{X}:\vec{k}}\, A.\, S \triangleq \mathbf{!}\,(\vec{X} : \vec{\mathsf{Type}}_k)(v : \mathsf{Val})\,\langle v\rangle\{A\,v\}.\, S$$
$$\mathbf{?}_{\vec{X}:\vec{k}}\, A.\, S \triangleq \mathbf{?}(\vec{X} : \vec{\mathsf{Type}}_k)(v : \mathsf{Val})\,\langle v\rangle\{A\,v\}.\, S$$

The binders $\vec{X}$ are kinded so that we can quantify over both term types and session types.

This definition relies on the fact that binders $\vec{x}{:}\vec{\tau}$ in Actris's dependent separation protocols $\mathbf{!}\,\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\,prot$ and $\mathbf{?}\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\,prot$ are higher-order and impredicative (*i.e.,* they allow quantification over Iris predicates and Actris protocols). The typing rules are extended to allow instantiation of binders when sending a message, and elimination of type variables when receiving a message. Concretely, the rule for channel creation remains unchanged, while the rules for send and receive become:

$$\frac{\Gamma \vDash e : A[\vec{K}/\vec{X}] \dashv \Gamma', x : \texttt{chan}\ (!_{\vec{X}:\vec{k}}\, A.\, S)}{\Gamma \vDash \texttt{send}\ x\ e : \mathbf{1} \dashv \Gamma', x : \texttt{chan}\ S[\vec{K}/\vec{X}]}$$

$$\frac{\Gamma, x : \texttt{chan}\ S, y : A \vDash e : B \dashv \Gamma' \qquad \vec{X} \notin FV(\Gamma, \Gamma', B)}{\Gamma, x : \texttt{chan}\ (?_{\vec{X}:\vec{k}}\, A.\, S) \vDash \texttt{let}\ y := \texttt{recv}\ x\ \texttt{in}\ e : B \dashv \Gamma' \setminus \{y\}}$$

The second rule requires the result y of `recv` to be let-bound to ensure that the type variables $\vec{X}$ cannot escape into the context Γ' or the type B.

With this rule, we can type check the following function that follows the computation service type $\overline{\texttt{compute\_type}}$:

```
compute_service ≜ rec go c :=
  branch c with
    left  ⇒ let f := recv c in send (f ()) ; go c
  | right ⇒ ()
  end
```

We can prove the typing judgement $\Gamma \vDash \texttt{compute\_service} : \texttt{chan}\ \overline{\texttt{compute\_type}} \to () \dashv \Gamma$ using only the typing rules of our semantic type system. In section 4.4.2 we consider a client that uses this service, which cannot itself be type checked using our typing rules but rather requires a manual proof of its typing judgement.

4.3.6 Locks and Mutexes

The substructural nature of channels (of type `chan` S) ensure that they can be used by at most one thread at the same time. Balzer and Pfenning [2017] proposed a more liberal extension of session types that allows channels to be shared between multiple threads via locks. We show that we can achieve a similar kind of sharing by extending our type system with a type former `mutex` A of mutexes (*i.e.,* lock-protected values of type A) inspired by Rust's Mutex library. For example, mutexes make it possible to share the channel to the computation service `compute_type` from Section 4.3.5 between multiple clients—they can acquire the mutex (`mutex compute_type`), send any number of computation requests, retrieve the corresponding results, and then release the mutex.

The `mutex` type former is copyable, and comes with operations `newmutex` to allocate a mutex, `acquiremutex` to acquire a mutex by blocking until no other thread holds it, and `releasemutex` to release the mutex. The typing rules are shown in Figure 4.4 and include the type former $\overline{\texttt{mutex}}$, which signifies that the mutex is acquired.

Iris's proof rules for locks:

$$\begin{aligned}
\mathsf{is_lock}\ lk\ R &\twoheadrightarrow \square(\mathsf{is_lock}\ lk\ R)\\
R &\twoheadrightarrow \mathsf{wp}\ \texttt{new_lock}\ ()\ \{lk.\ \mathsf{is_lock}\ lk\ R\}\\
\mathsf{is_lock}\ lk\ R &\twoheadrightarrow \mathsf{wp}\ \texttt{acquire}\ lk\ \{R\}\\
\mathsf{is_lock}\ lk\ R * R &\twoheadrightarrow \mathsf{wp}\ \texttt{release}\ lk\ \{\mathsf{True}\}
\end{aligned}$$

Semantic typing rules for mutexes:

$$\texttt{copyable}\,(\texttt{mutex}\,A)$$

$$\Gamma \vDash \texttt{newmutex} : A \to \texttt{mutex}\,A \dashv \Gamma$$

$$\Gamma, x : \texttt{mutex}\,A \vDash \texttt{acquiremutex}\ x : A \dashv \Gamma, x : \overline{\texttt{mutex}}\,A$$

$$\frac{\Gamma \vDash e : A \dashv \Gamma', x : \overline{\texttt{mutex}}\,A}{\Gamma \vDash \texttt{releasemutex}\ x\ e : \mathbf{1} \dashv \Gamma', x : \texttt{mutex}\,A}$$

Figure 4.4: Iris's proof rules for locks and semantic typing rules for mutexes.

To extend our type system with mutexes we make use of the locks library that is available in Iris. This library consists of operations `newlock`, `acquire`, and `release`, which are similar to the mutex operations, but do not protect a value. The mutex operations are defined in terms of locks as follows:

$$\begin{aligned}
\texttt{newmutex} &\triangleq \lambda y.\ (\texttt{new_lock}\ (), \texttt{ref}\ y)\\
\texttt{acquiremutex} &\triangleq \lambda x.\ \texttt{acquire}\ (\texttt{fst}\ x);\ !\,(\texttt{snd}\ x)\\
\texttt{releasemutex} &\triangleq \lambda x\ y.\ (\texttt{snd}\ x) \leftarrow y;\ \texttt{release}\ (\texttt{fst}\ x)
\end{aligned}$$

That is, `newmutex` creates a lock alongside a boxed value. The value can then be acquired with `acquiremutex`, which first acquires the lock. Finally, `releasemutex` moves the value back into the box, and releases the lock.

The Iris rules for locks are shown in Figure 4.4 and make use of the representation predicate $\mathsf{is\_lock}\ lk\ R$, which expresses that a lock lk guards the resources R. When creating a new lock one has to give up ownership of R, and in turn, obtains the representation predicate $\mathsf{is\_lock}\ lk\ R$. The representation is persistent, so it can be freely duplicated. When entering a critical section using $\texttt{acquire}\ lk$, a thread gets exclusive ownership of R, which has to be given up when releasing the lock using $\texttt{release}\ lk$. Using the lock representation predicate, we define type formers for mutexes:

$$\begin{aligned}
&\texttt{mutex}\,A \triangleq \lambda w.\ \exists lk, \ell.\ (w = (lk, \ell)) * \mathsf{is_lock}\ lk\ (\exists v.\ (\ell \mapsto v) * \triangleright(A\,v))\\
&\overline{\texttt{mutex}}\,A \triangleq \lambda w.\ \exists lk, \ell.\ (w = (lk, \ell)) * \mathsf{is_lock}\ lk\ (\exists v.\ (\ell \mapsto v) * \triangleright(A\,v)) * (\ell \mapsto -)
\end{aligned}$$

The `mutex` type former states that its values are pairs of locks and boxed values. The $\overline{\texttt{mutex}}$ type former additionally asserts ownership of the reference, implying that the lock has been acquired. The typing rules for mutexes as shown in Figure 4.4 are proven as lemmas.

4.3.7 Session-Level Subtyping

Session-level subtyping $S <: T$, originally presented by Gay and Hole [2005], relates session subtypes S with session supertypes T, that can be used in place of the subtype, captured by monotonicity with the subtyping of the channel type:

$$\frac{S <: T}{\texttt{chan}\ S <: \texttt{chan}\ T}$$

Subtyping in session types allows sending supertypes and receiving subtypes, as well as increasing and reducing the range of choices for branchings and selections, respectively:

$$\frac{A_2 <: A_1 \quad S_1 <: S_2}{!A_1.\, S_1 <: !A_2.\, S_2} \qquad \frac{A_1 <: A_2 \quad S_1 <: S_2}{?A_1.\, S_1 <: ?A_2.\, S_2} \qquad \frac{\vec{S}_2 \subseteq \vec{S}_1}{\oplus\{\vec{S}_1\} <: \oplus\{\vec{S}_2\}} \qquad \frac{\vec{S}_1 \subseteq \vec{S}_2}{\&\{\vec{S}_1\} <: \&\{\vec{S}_2\}}$$

This is essential for program reuse, *e.g.,* any program that handles more choices than indicated by a branch type should be able to accept a channel with that branch type.

In asynchronous session types, one can further extend subtyping with a "swapping" rule $?A_1.\, !A_2.\, S <: !A_2.\, ?A_1.\, S$ that allows performing sends (**!**) ahead of receives (**?**), and similar rules that allow performing selects ($\oplus$) ahead of receives (**?**), sends (**!**) ahead of branches (&), and selects ($\oplus$) ahead of branches (&) [Mostrous et al. 2009][2]. For example, using swapping, a client of the computation service from Section 4.3.5, with type `compute_type` can swap the selects and sends ahead of receives, to send multiple computation requests at once, and only then await the computed results.

To extend our semantic session types with session subtyping, we make use of Actris's notion of *subprotocols* [Hinrichsen et al. 2021a], for which the rules are shown in Figure 4.5. The first four rules mimic the behaviour of session subtyping in how it is possible to send more and receive less, while accounting for the protocol-level binders of dependent separation protocols. In particular, we can (1) move out binders and propositions of right-hand side sending protocols ($\sqsubseteq$-SEND-OUT) and (2) left-hand side receiving protocols ($\sqsubseteq$-RECV-OUT), and (3) move in binders and propositions of right-hand side sending protocols ($\sqsubseteq$-SEND-IN) and (4) left-hand side receiving protocols ($\sqsubseteq$-RECV-IN). Rule $\sqsubseteq$-SWAP accounts for the swapping of sends and receives that are independent of each other, as guaranteed by the omission of binders in the rule. If binders are present, the first four rules should be used first. Rules $\sqsubseteq$-SEND-MONO and $\sqsubseteq$-RECV-MONO account for the monotonicity of the subprotocol relation in the tails,

[2]Discrepancies in the direction between the swapping rules of Mostrous et al. [2009] and us will be discussed in Section 4.6.

Actris's proof rules for subprotocols:

$$\begin{aligned}
\left(\forall \vec{x}{:}\vec{\tau}.\, P \mathrel{-\!\!*} \left(prot_1 \sqsubseteq \mathbf{!}\,\langle v\rangle.\, prot_2\right)\right) &\mathrel{-\!\!*} \left(prot_1 \sqsubseteq \mathbf{!}\,\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\, prot_2\right) && (\sqsubseteq\text{-SEND-OUT})\\
\left(\forall \vec{x}{:}\vec{\tau}.\, P \mathrel{-\!\!*} \left(\mathbf{?}\langle v\rangle.\, prot_1 \sqsubseteq prot_2\right)\right) &\mathrel{-\!\!*} \left(\mathbf{?}\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\, prot_1 \sqsubseteq prot_2\right) && (\sqsubseteq\text{-RECV-OUT})\\
P[\vec{t}/\vec{x}] \mathrel{-\!\!*} \left(\mathbf{!}\,\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\, prot\right) &\sqsubseteq \mathbf{!}\,\langle v[\vec{t}/\vec{x}]\rangle.\, prot[\vec{t}/\vec{x}]) && (\sqsubseteq\text{-SEND-IN})\\
P[\vec{t}/\vec{x}] \mathrel{-\!\!*} \left(\mathbf{?}\langle v[\vec{t}/\vec{x}]\rangle.\, prot[\vec{t}/\vec{x}]\right. &\sqsubseteq \mathbf{?}\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\, prot) && (\sqsubseteq\text{-RECV-IN})\\
\mathbf{?}\langle v_1\rangle\{P_1\}.\, \mathbf{!}\,\langle v_2\rangle\{P_2\}.\, prot &\sqsubseteq \mathbf{!}\,\langle v_2\rangle\{P_2\}.\, \mathbf{?}\langle v_1\rangle\{P_1\}.\, prot && (\sqsubseteq\text{-SWAP})\\
\triangleright(prot_1 \sqsubseteq prot_2) \mathrel{-\!\!*} \left(\mathbf{!}\,\langle v\rangle\{P\}.\, prot_1\right. &\sqsubseteq \mathbf{!}\,\langle v\rangle\{P\}.\, prot_2) && (\sqsubseteq\text{-SEND-MONO})\\
\triangleright(prot_1 \sqsubseteq prot_2) \mathrel{-\!\!*} \left(\mathbf{?}\langle v\rangle\{P\}.\, prot_1\right. &\sqsubseteq \mathbf{?}\langle v\rangle\{P\}.\, prot_2) && (\sqsubseteq\text{-RECV-MONO})\\
(prot_1 \sqsubseteq prot_2) \mathrel{-\!\!*} (c \rightarrowtail prot_1) &\mathrel{-\!\!*} (c \rightarrowtail prot_2) && (\sqsubseteq\text{-CHAN-MONO})
\end{aligned}$$

Semantic subtyping rules for session polymorphism:

$$\frac{S_1 <: \mathbf{!}A.\, S_2}{S_1 <: \mathbf{!}_{(\vec{X}:\vec{k})} A.\, S_2} \qquad \frac{\mathbf{?}A.\, S_1 <: S_2}{\mathbf{?}_{(\vec{X}:\vec{k})} A.\, S_1 <: S_2}$$

$$\mathbf{!}_{(\vec{X}:\vec{k})}\, A.\, S <: \mathbf{!}A[\vec{K}/\vec{X}].\, S[\vec{K}/\vec{X}] \qquad \mathbf{?}A[\vec{K}/\vec{X}].\, S[\vec{K}/\vec{X}] <: \mathbf{?}_{(\vec{X}:\vec{k})}\, A.\, S$$

Figure 4.5: A selection of the Actris's proof rules for subprotocols and semantic session subtyping rules. $\sqsubseteq$-SEND-OUT and $\sqsubseteq$-RECV-OUT only hold if $\tau \in \vec{\tau}$ is inhabited

and rule $\sqsubseteq$-CHAN-MONO states that Actris's connective for channel ownership is closed under the subprotocol relation. The subprotocol relation is reflexive and transitive.

With Actris's subprotocol relation at hand, we define the semantic subtyping relation for session types as follows:

$$S <: T \triangleq S \sqsubseteq T$$

We then prove the conventional subtyping rules for asynchronous session types as lemmas using the rules in Figure 4.5 for Actris's subprotocol relation. These subtyping rules include, but are not limited to, contra- and covariance of the type A of the send $\mathbf{!}A.\, S$ and receive $\mathbf{?}A.\, S$ session types respectively, the various forms of swapping as described in the beginning of this section, and the rules for reducing and increasing the range of choices for selecting and branching protocols as also shown in the beginning of this section.

As a new feature, which up to our knowledge is not present in existing session type systems, we prove the subtyping rules for polymorphic session types as shown in Figure 4.5. For sending session types, we can instantiate the polymorphic types of subtypes, and generalise over the polymorphic types for supertypes. Conversely, for receiving session types, we can instantiate the polymorphic types of supertypes, and generalise over the polymorphic types for subtypes.

Subtyping for polymorphic session types is useful to describe the interaction between generic services and concrete clients. For example, consider a mapping service

to which one can send a function $A \multimap B$, a value A, and get back the mapped result B. The most generic session type for interacting with such a service would be the following:

$$!_{(X,Y:\bigstar)}\,(X \multimap Y).\,!X.\,?Y.\,\mathtt{end}$$

Now assume that we have type checked a concrete client with the following session type:

$$!(\mathbf{Z} \multimap \mathbf{B}).\,!\mathbf{Z}.\,?\mathbf{B}.\,\mathtt{end}$$

While this concrete session type is not compatible with the one expected by the service, we can use the subtyping relation to weaken the generic type into the concrete one:

$$!_{(X,Y:\bigstar)}\,(X \multimap Y).\,!X.\,?Y.\,\mathtt{end} \sqsubseteq\ !(\mathbf{Z} \multimap \mathbf{B}).\,!\mathbf{Z}.\,?\mathbf{B}.\,\mathtt{end}$$

The judgement follows from $!_{(\vec{X}:\vec{k})}\,A.\,S <:\ !A[\vec{K}/\vec{X}].\,S[\vec{K}/\vec{X}]$. Conversely, one could allocate the types from the perspective of the concrete client, and then weaken the service type into the generic type, by generalising over the received types:

$$?(\mathbf{Z} \multimap \mathbf{B}).\,?\mathbf{Z}.\,!\mathbf{B}.\,\mathtt{end} \sqsubseteq\ ?_{(X,Y:\bigstar)}\,(X \multimap Y).\,?X.\,!Y.\,\mathtt{end}$$

This subtyping judgement follows from the rule for receive $?A[\vec{K}/\vec{X}].\,S[\vec{K}/\vec{X}] <:\ ?_{(\vec{X}:\vec{k})}\,A.\,S$.

4.4 Manual Typing Proofs

We demonstrate how safe programs that are not typeable using the existing typing rules can be assigned a typing judgement via a manual proof in Iris/Actris. We call such proofs *manual typing proofs*. As advocated by Jung et al. [2018a, 2021], such proofs are useful since typing judgements, regardless of whether they have been derived manually or by using our typing rules, are interchangeable. While Jung et al. use such proofs to verify low-level concurrent libraries, we use them to verify binary message-passing programs where the user of one endpoint is verified using existing typing rules, and the other via a manual typing proof.

We first provide an intuition for the manual typing proof approach by proving the typing judgement of the parallel receiving program from the introduction (Section 4.4.1), and then show a more realistic example by proving the typing judgement of a parallel client of the computation service from Section 4.3.5 that uses a producer/consumer pattern (Section 4.4.2).

4.4.1 Receiving in Parallel

Consider the example from Section 4.1 (where the locks have been made explicit):

```
threadprog ≜
  λc lk. acquire lk;
         let x := recv c in
         release lk;
         x

lockprog ≜
  λc. let lk := new_lock () in
      (threadprog c lk || threadprog c lk)
```

We want to prove the following typing judgement:

$$\Gamma \vDash \texttt{lockprog} : \texttt{chan}\ (\mathbf{?Z}.\,\mathbf{?Z}.\,\mathbf{end}) \multimap (\mathbf{Z} \times \mathbf{Z}) \dashv \Gamma$$

This typing judgement is not derivable from the typing rules we presented so far, even with mutexes instead of plain locks, as the channel type changes each time the lock/mutex is acquired and released. However, we can unfold the definition of the semantic typing judgement and types, which gives us the following proof obligation in Iris/Actris:

$$(c \rightarrowtail \mathbf{?}(v_1 : \mathsf{Val})\,\langle v_1\rangle\{v_1 \in \mathbb{Z}\}.\,\mathbf{?}(v_2 : \mathsf{Val})\,\langle v_2\rangle\{v_2 \in \mathbb{Z}\}.\,\mathbf{end}) \mathrel{-\!\!*}$$
$$\mathsf{wp}\ \texttt{lockprog}\ c \left\{ v.\, \exists v_1, v_2.\, \begin{array}{l} (v = (v_1, v_2)) * \\ \triangleright(v_1 \in \mathbb{Z}) * \triangleright(v_2 \in \mathbb{Z}) \end{array} \right\}$$

The proof of above obligation is carried out using Iris's support for fractional permissions $\boxed{q}^{\gamma}$ where $q \in (0, 1]_{\mathbb{Q}}$ and γ is an identifier. The permission reflects how much of the channel protocol its owner is allowed to resolve, enforced by the following lock invariant:

$$\begin{array}{lll} \texttt{chaninv}\ c \triangleq & & \\ \quad (c \rightarrowtail \mathbf{?}(v_1 : \mathsf{Val})\,\langle v_1\rangle\{v_1 \in \mathbb{Z}\}.\,\mathbf{?}(v_2 : \mathsf{Val})\,\langle v_2\rangle\{v_2 \in \mathbb{Z}\}.\,\mathbf{end}) & \vee & (i) \\ \quad (c \rightarrowtail \mathbf{?}(v_2 : \mathsf{Val})\,\langle v_2\rangle\{v_2 \in \mathbb{Z}\}.\,\mathbf{end} * \boxed{1/2}^{\gamma}) & \vee & (ii) \\ \quad (c \rightarrowtail \mathbf{end} * \boxed{1}^{\gamma}) & & (iii) \end{array}$$

The invariant describes that the channel is in one of three states: (i) no values have been received yet, (ii) one value has been received, or (iii) all values have been received. State (ii) and (iii) assert that the invariant (not the thread) has half and full ownership of the fractional permission respectively.

The proof is carried out by allocating a full fractional permission $\boxed{1}^{\gamma}$ (with a fresh identifier γ), after which the lock predicate $\mathsf{is\_lock}\ lk\ (\texttt{chaninv}\ c)$ is allocated by giving up ownership of the channel c, where $\texttt{chaninv}\ c$ is initially in state (i). The fractional permission is then split into two halves $\boxed{1/2}^{\gamma}$, which are each delegated to a thread, along with the persistent lock predicate $\mathsf{is\_lock}\ lk\ (\texttt{chaninv}\ c)$. Both threads have the same proof obligation:

$$(\mathsf{is\_lock}\ lk\ (\texttt{chaninv}\ c) * \boxed{1/2}^{\gamma}) \mathrel{-\!\!*} \mathsf{wp}\ \texttt{threadprog}\ c\ lk\ \{v.\, v \in \mathbb{Z}\}$$

First, the lock invariant is obtained by acquiring the lock. The channel can then either be in state (i) or (ii), as having half of the fractional permission excludes the possibility of the full fraction being in the lock (and thereby state (iii)).

If the invariant is in state (i), the thread takes a step of the protocol and surrenders its fractional permission $\boxed{1/2}^{\gamma}$ leaving the invariant in state (ii); if the invariant is in state (ii) a similar step is taken leaving the invariant with the full fractional permission $\boxed{1}^{\gamma}$ in state (iii).

4.4.2 A Parallel Computation Client

In section 4.3.5 we considered the session type `compute_type` for a client of a polymorphic recursive computation service. We now consider a client `compute_client`,

```
compute_client ≜ λl c.
  let n := llength l in
  let ctr := ref 0 in
  let l' := lnil () in
  let lk := new_lock () in
  (produce l ctr lk c ||
   consume l' n ctr lk c);
  l'
```

```
produce ≜
  rec go l ctr lk c :=
    if lisnil l then
      acquire lk;
        select c stop;
      release lk
    else
      acquire lk;
        select c cont;
        send c (lpop l);
        ctr ← ! ctr + 1
      release lk;
      go l ctr lk c
```

```
consume ≜
  rec go l n ctr lk c :=
    if n = 0 then () else
    acquire lk;
    if ! ctr = 0 then
      release lk;
      go l n ctr lk c
    else
      let x := recv c in
      ctr ← ! ctr − 1;
      release lk;
      go l (n − 1) ctr lk c;
      lcons x l
```

Figure 4.6: A producer-consumer client for the computation service. (The operations on lists `llength`, `lnil`, `lisnil`, and `lpop`, are standard and their code have thus been elided).

shown in Figure 4.6, which interacts with the service by sending a list of computation requests and receiving their results in parallel, similar to the producer-consumer pattern.[3] We want to prove:

$$\Gamma \vDash \texttt{compute\_client} : \texttt{list}\ (\mathbf{1} \multimap A) \multimap \texttt{chan}\ \texttt{compute\_type} \multimap \texttt{list}\ A\ \dashv\ \Gamma$$

where $\texttt{list}\ A \triangleq \mu rec.\, \texttt{ref}_{\texttt{uniq}}\,(\mathbf{1} + (A \times rec))$.

The client `compute_client` operates on a channel endpoint c, where the computation service has the other endpoint. The client creates a shared counter ctr to keep track of the number of requests that are being processed, a linked list l' for the results, and a lock lk. It runs the producer `produce` and consumer `consume` in parallel, which both race for the lock lk to access the channel c and counter ctr. The producer processes the input list l one-by-one by sending each computation in l on the channel c, and increasing the shared counter ctr thereafter. The consumer `consume` adds the results one-by-one to the list l' by receiving them on the channel c, and decreasing the shared counter ctr thereafter. When both the producer and consumer terminate, the client returns the list l' that then contains the results.

The type system cannot type check `compute_client`, as (1) its safety depends on the length of the list, which is not available from the type, and (2) the channel c is shared and the type changes between each concurrent access. To prove that `compute_client` is semantically typed, we unfold its typing judgement, and resolve each step of the program in sequence, by applying the related weakest precondition

[3]For simplicity, our producer and consumer just iterate through a list, whereas in reality they would perform some computations so there is a point in having the producer and consumer operate in parallel.

rules. We first use the weakest precondition rule for `llength`:

$$\texttt{list}\ A\ l \mathrel{-\!\!*} \texttt{wp llength}\ l\left\{n.\, n = |\vec{v}| * l \overset{\text{list}}{\mapsto}_A \vec{v}\right\}$$

This rule converts the type predicate $\texttt{list}\ A$ of the linked list l into the separation-logic list representation predicate $l \overset{\text{list}}{\mapsto}_A \vec{v}$, which additionally makes the contents $\vec{v}$ of the linked list l explicit. This predicate is defined as follows:

$$l \overset{\text{list}}{\mapsto}_A \vec{v} \triangleq \begin{cases} l \mapsto \texttt{inl}\ () & \text{if } \vec{v} = [\,] \\ \exists l_2.\, l \mapsto \texttt{inr}\ (v_1, l_2) * A\ v_1 * l_2 \overset{\text{list}}{\mapsto}_A \vec{v}_2 & \text{if } \vec{v} = [v_1] \cdot \vec{v}_2 \end{cases}$$

The remainder of the proof is similar to the proof of the parallel receive in Section 4.4.1—we establish a lock invariant $\texttt{chaninv}_{pc}\ ctr\ c\ A$ to share the counter ctr and the channel c between the producer and consumer, and use a fractional permission $\boxed{q}^{\gamma}$ to determine the state of the shared channel c:

$$\begin{aligned} \texttt{chaninv}_{pc}\ ctr\ c\ A \triangleq \exists n.\, ctr \mapsto n * \\ (c \rightarrowtail ((\texttt{?}A)^n \cdot \texttt{compute\_type}) \lor & \quad (i) \\ (c \rightarrowtail ((\texttt{?}A)^n \cdot \texttt{end}) * \boxed{1}^{\gamma})) & \quad (ii) \end{aligned}$$

The lock invariant states that the session type of the channel starts with a sequence of receive actions $(\texttt{?}A)^n$, where n is the value of the shared counter ctr. Here, the notation S^n denotes S appended to itself n times (the append operation $\cdot$ is inherited from Actris). The invariant expresses that either (i) the channel is still open, which permits unfolding the recursive definition to send additional requests, or (ii) the channel terminates with `end`, after the n receive steps have been resolved. State (ii) requires the full fractional permission $\boxed{1}^{\gamma}$, which must be released before closing the channel.

The proof is carried out by allocating the fractional permission $\boxed{1}^{\gamma}$ (with a fresh identifier γ), after which the weakest precondition rules for parallel composition (see Figure 4.2), the producer `produce`, and consumer `consume` are used:

$$\begin{aligned} &\texttt{is\_lock}\ lk\ (\texttt{chaninv}_{pc}\ ctr\ c\ A) * \boxed{1}^{\gamma} * l \overset{\text{list}}{\mapsto}_{(\mathbf{1} \multimap A)} \vec{v}\ \mathrel{-\!\!*} \\ &\qquad \texttt{wp produce}\ l\ ctr\ lk\ c\left\{l \overset{\text{list}}{\mapsto}_{(\mathbf{1} \multimap A)} [\,]\right\} \\ &\texttt{is\_lock}\ lk\ (\texttt{chaninv}_{pc}\ ctr\ c\ A) * l \overset{\text{list}}{\mapsto}_A [\,]\ \mathrel{-\!\!*} \\ &\qquad \texttt{wp consume}\ l\ n\ ctr\ lk\ c\left\{\exists \vec{w}.\, |\vec{w}| = n * l \overset{\text{list}}{\mapsto}_A \vec{w}\right\} \end{aligned}$$

The proof of `produce` proceeds as follows. Owning $\boxed{1}^{\gamma}$ means the lock invariant is in state (i). Therefore, after unfolding the recursive tail and instantiating the polymorphic binder in the type of c, we have:

$$c \rightarrowtail (\texttt{?}A)^n \cdot \oplus \left\{\begin{array}{l} \texttt{cont} : \texttt{!}(\mathbf{1} \multimap A).\, \texttt{?}A.\, \texttt{compute\_type} \\ \texttt{stop} : \texttt{end} \end{array}\right\}$$

The select and send actions can then be swapped ahead of the receives, resulting in:

$$c \rightarrowtail \oplus \left\{\begin{array}{l} \texttt{cont} : \texttt{!}(\mathbf{1} \multimap A).\, (\texttt{?}A)^{n+1} \cdot \texttt{compute\_type} \\ \texttt{stop} : (\texttt{?}A)^n \cdot \texttt{end} \end{array}\right\}$$

If the list is non-empty, we resolve a computation step (by selecting the `cont` branch and sending a computation with type $\mathbf{1} \multimap A$) resulting in $c \rightarrowtail (?A)^{n+1} \cdot \texttt{compute\_type}$. After incrementing the shared counter *ctr*, we reestablish the lock invariant in state (i). If the list is empty, we close the channel (by selecting the `stop` branch), resulting in $c \rightarrowtail (?A)^n \cdot \texttt{end}$. We reestablish the lock invariant in state (ii) by giving up the fractional permission $\ulcorner \overline{1} \urcorner^{\gamma}$.

The proof of `consume` proceeds as follows. We only perform a receive operation when the shared counter *ctr* is positive, which means we have $c \rightarrowtail ?A.\, (?A)^{n-1} \cdot S$. Here, S is `compute_type` or `end`, depending on whether the lock invariant is in state (i) or (ii), respectively. After the receive operation we have $c \rightarrowtail (?A)^{n-1} \cdot S$, so after decrementing the shared counter *ctr* we can reestablish the lock invariant.

To finalise the proof of the client `compute_client`, we weaken $l \overset{\text{list}}{\mapsto}_A \vec{w}$ returned by `consume` to `list` $A\ l$ by forgetting about the contents $\vec{w}$ of the linked list l.

4.5 Mechanisation in Coq

In this paper, we have used what is often called the "foundational approach" to semantic type safety [Appel and McAllester 2001; Ahmed 2004; Ahmed et al. 2010]. That means that contrary to conventional logical relation developments, types are not defined syntactically, and then given a semantic interpretation. Instead, types are defined as combinators in terms of their semantic interpretation. This approach gives rise to an "open" system that can easily be extended with new type formers, and is thus particularly suitable for mechanisation in a proof assistant like Coq. Furthermore, as we will show in this section, the foundational approach makes it possible to reuse Coq's variables to model type-level binding, avoiding boilerplate that would be necessary with a first-order representation of variable binding.

Our mechanisation is built on top of the mechanisation of Iris and Actris in Coq, which provides a number of noteworthy advantages. First, we can reuse their libraries for various programming constructs, such as locks (from Iris) and channels (from Actris). Second, we avoid reasoning about explicit resources in Coq by making use of the MoSeL framework (formerly, Iris Proof Mode), which provides tactics tailored for reasoning about the connectives of separation logic, and hides unnecessary details related to the embedding of separation logic in Coq [Krebbers et al. 2017b, 2018].

Typing Judgments Term and session types are represented as a dependent type indexed by a kind:[4]

```
Inductive kind := tty_kind | sty_kind. (* ★ or ♦ *)

Inductive lty Σ : kind → Type :=
  | Ltty : (val → iProp Σ) → lty Σ tty_kind
  | Lsty : iProto Σ → lty Σ sty_kind.
Notation ltty Σ := (lty Σ tty_kind).
Notation lsty Σ := (lty Σ sty_kind).
```

[4] As is common in Iris, all definitions are parameterised by a Σ, which describes the resources that are available. For the purpose of this paper, this technicality can be ignored.

Typing contexts are represented as association lists:

```
Inductive ctx_item Σ := CtxItem {
  ctx_item_name : string;
  ctx_item_type : ltty Σ }.
Notation ctx Σ := (list (ctx_item Σ)).
```

The semantic term typing judgement is defined as:

```
(* ltty_car: ltty Σ → (val → iProp Σ) is the
   inverse of Ltty *)
Definition ltyped (Γ1 Γ2 : ctx Σ)
    (e : expr) (A : ltty Σ) : iProp Σ :=
  ■ ∀ vs, ctx_ltyped vs Γ1 -∗
       WP subst_map vs e {{ v, ltty_car A v ∗ ctx_ltyped vs Γ2 }}.
Notation "Γ1 ⊨ e : A ⫤ Γ2" :=
  (ltyped Γ1 Γ2 e A) : bi_scope.
Notation "Γ1 ⊨ e : A ⫤ Γ2" :=
  (⊢ ltyped Γ1 Γ2 e A) : type_scope.
```

The typing judgement is defined for the deeply-embedded expressions `expr` of the (untyped) language HeapLang, which is the default language shipped with Iris, and is extended by the Actris framework with connectives for message passing. HeapLang use strings for variables, and hence our typing contexts `ctx` do that too. Compared to *e.g.,* De Bruijn indices or locally nameless, the use of strings makes it possible to write programs in a human-readable way.[5]

The typing judgement is identical to the definition in Section 4.2.3, but is defined as an internal notion in Iris, *i.e.,* it is an Iris proposition `iProp` instead of a Coq proposition `Prop`. This provides some flexibility in manual typing proofs. For example, it makes it possible to prove typing judgements using Löb induction, without having to unfold their definition. To ensure that the typing judgement can be used as an ordinary proposition of higher-logic in Iris, it contains the *plainly modality* (■), which ensures that it does not capture any separation logic resources.[6] We define two notations so the typing judgement can be used internally and externally. The second notation uses the validity predicate of Iris (⊢), which turns an `iProp` into a `Prop`.

Typing Lemmas As an example of how a semantic typing rule looks like in Coq, consider the lemma corresponding to the typing rule for let-expressions:

```
Lemma ltyped_let Γ1 Γ2 Γ3 x e1 e2 A1 A2 :
  (Γ1 ⊨ e1 : A1 ⫤ Γ2) -∗
  (ctx_cons x A1 Γ2 ⊨ e2: A2 ⫤ Γ3) -∗
  (Γ1 ⊨ (let: x := e1 in e2) : A2 ⫤ ctx_filter_eq x Γ2 ++ ctx_filter_ne x Γ3).
```

The typing rule of let shows the handling of shadowing of variables: `ctx_cons x A1 Γ2` removes all bindings of `x` from `Γ2` before adding the new binding, and `ctx_filter_eq x Γ2` makes sure that potentially overshadowed variables are preserved. Dealing with

[5] Since HeapLang's operational semantics is defined on closed terms, the use of strings does not cause issues with variable capture. See also [Pierce et al. 2020, Section STLC] for a discussion on the use of strings for variables.

[6] The plainly modality (■) is like the persistent modality (□), but additionally makes sure no persistent resources are captured.

shadowing in the proof is trivial due to some general-purpose lemmas for $\Gamma \vDash \sigma$ (`ctx_ltyped Γ vs` in Coq). The proof of the typing rule is 9 lines of Coq code.

The term type for kinded universal types is defined as:

```
Definition lty_forall {k}
    (C : lty Σ k → ltty Σ) : ltty Σ :=
  Ltty (λ w, ∀ X, WP w #() {{ ltty_car (C X) }}).
Notation "∀ X, C" := (lty_forall (λ X, C)): lty_scope.

Lemma ltyped_tlam Γ1 Γ2 Γ' e k (C : lty Σ k → ltty Σ) :
  (∀ K, Γ1 ⊨ e: C K ⊣ []) -∗
  (Γ1 ++ Γ2 ⊨ (λ: <>, e) : (∀ X, C X) ⊣ Γ2).
Lemma ltyped_tapp Γ Γ2 e k (C : lty Σ k → ltty Σ) K :
  (Γ ⊨ e : (∀ X, C X) ⊣ Γ2) -∗
  (Γ ⊨ e #() : C K ⊣ Γ2).
```

The universal type shows how the semantic approach allows binders to be modelled using Coq's binders. The argument `C` of `lty_forall` is a Coq function, and thus the binding in the notation `∀X, C` is simply achieved using a Coq lambda abstraction `λX, C`. This approach gives the same feeling of working with higher-order abstract syntax [Pfenning and Elliott 1988], albeit being semantical instead of syntactical. The typing rule for type abstraction similarly uses Coq's binders, where the `∀ K` in the premise implicitly ensures that `K` is fresh. The proof of the two typing rules are 4 and 3 lines of code, respectively.

The session type for selection ($\oplus$) and branching (&) is:

```
Inductive action := Send | Recv.
Definition lty_choice (a : action)
    (Ss : gmap Z (lsty Σ)) : lsty Σ :=
  Lsty (<a@(i: Z)> MSG #x {{ ⌜is_Some (Ss !! i)⌝ }}; lsty_car (Ss !!! i)).
Notation lty_select := (lty_choice Send).

Lemma ltyped_select Γ x i S Ss :
  Γ !! x = Some (chan (lty_select Ss)) →
  Ss !! i = Some S →
  Γ ⊨ select x #i : () ⊣ env_cons x (chan S) Γ.
```

Since $\oplus$ and & are dual, this definition (as well as in many other dual definitions, lemmas, and proofs) are factorised using the inductive type `action`. The syntax `<a@(`$\vec{x}$:$\vec{\tau}$`)> MSG` v `{{` P `}};` *prot* expands to Actris's $!\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\,prot$ or $?\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\,prot$ depending on the action `a`. The definition uses the finite map library `gmap` of std++ [The Coq-std++ Team 2020], to represent the choices `Ss`. The notation `Ss !!! i` is the lookup function on maps, whose result is only well-defined if `i` is in the map `Ss`, as required by `is_Some (Ss !! i)`. The notation `⌜_⌝` embeds a Coq `Prop` into Iris. The typing rule for select requires the label `i` to be in the map `Ss`, and updates the channel `S` based on the label. The proof uses Actris's proof rules, and is 6 lines of code.

Type Safety. The type safety lemma is stated as follows:

```
Lemma ltyped_safety e σ es σ' e' :
  (∃ A, [] ⊨ e : A ⊣ []) →
  rtc erased_step ([e], σ) (es, σ') → e' ∈ es →
  is_Some (to_val e') ∨ reducible e' σ'.
```

This lemma states that if we have a typing judgment for a closed expression `e`, and we start execution of the single thread `e` to obtain a list of resulting threads `es` after any number of execution steps (modeled using the reflexive-transitive closure, `rtc`, of HeapLang's small-step reduction relation), then any thread `e'` in `es` is either a value or can take a step.

4.6 Related Work

Session Types Seminal work on subtyping for binary recursive session types for a synchronous pi-calculus was done by Gay and Hole [2005]. Mostrous et al. [2009] expand on this work by adding support for multi-party asynchronous recursive session types, and later for higher-order process calculi [Mostrous and Yoshida 2015]. These two works present the session subtyping relation with inverted orientations, inverting the sub- and supertypes, which has been discussed by Gay [2016]. Our semantic session subtyping relation uses the same orientation as Gay and Hole. Mostrous et al. [2009] also present an output-input swapping rule, which inspired our swapping rule in Section 4.3.7, even though their type system is multi-party, as the idea is compatible with both session type variants. They additionally claim that their subtyping is decidable, it was later proven to not be the case by Bravetti et al. [2017], precisely because of the swapping rule.

Gay [2008] introduced bounded polymorphic session types where branches contain type variables for term types with upper and lower bounds. This work neither supports recursive types, session subtyping, nor delegation, but Gay hypothesised that recursion could be done. Dardha et al. [2012] expanded on this work by adding subtyping and delegation, while still only conjecturing that recursion was a possible extension. Caires et al. [2013] devised a polymorphic session type system for the synchronous pi-calculus with existential and universal quantifiers at the type-level, but not at the session-level. However, like Gay's work, their system supports neither recursive types nor subtyping.

Thiemann and Vasconcelos [2020] introduced label dependent session types, where tails can depend on the communicated message, which allows for encoding choice using send and receive. This is similar to the encoding of our semantic choice types in terms of Actris's dependent send and receive. While their work does not have asynchronous subtyping or polymorphism, it supports recursive types over natural numbers, with a recurser for type checking of such types.

Balzer and Pfenning [2017] and Balzer et al. [2019] proposed a session-type system that allows sharing of channels via locks. Their system contains unrestricted types that can be shared, linear types that cannot, and modalities to move between the two through the use of locks. Our mutex type works similarly with copyable types, but our system is more general, as the copyable types tie into Iris's general-purpose mechanisms for sharing. We can also impose mutexes on only one endpoint of a channel, while they require mutual locking on both ends, and integrate manual typing proofs of racy programs. They provide proofs for subject reduction and type preservation, not just to obtain type safety, but also to obtain deadlock freedom, which we do not consider.

Logical Relations Logical relations have been studied extensively in the context of Iris, for type safety of type systems [Krebbers et al. 2017b; Jung et al. 2018a; Giarrusso et al. 2020], program refinement [Krebbers et al. 2017b; Krogh-Jespersen et al. 2017; Tassarotti et al. 2017; Timany et al. 2018; Frumin et al. 2018], robust safety [Swasey et al. 2017], and non-interference [Frumin et al. 2020]. The most immediately related work in this area is the RustBelt project [Jung et al. 2018a], which uses logical relations to prove type safety and datarace-freedom of a large subset of Rust and its standard libraries, focusing on Rust's lifetime and borrowing mechanism. RustBelt employs the foundational approach to logical relations in its Coq development, from which we have drawn much inspiration. Giarrusso et al. [2020] used logical relations in Iris to prove type safety of a version of Scala's core calculus DOT, which has a rich notion of subtyping, but is different in nature from session subtyping.

The connection between logic and session types has been studied through the Curry-Howard correspondence by *e.g.,* Caires and Pfenning [2010], Wadler [2012], Carbone et al. [2017], and Dardha and Gay [2018]. As part of this line of work, Perez *et al.* used logical relations to prove termination [Pérez et al. 2012] and confluence [Pérez et al. 2014] of session-based concurrent systems.

Mechanisation of Session Types Mechanisations pertaining to session types are all fairly recent. There are two other mechanisations of session types in Iris. Tassarotti et al. [2017] proved termination preserving refinements for a compiler from a session-typed language to a functional language where message buffers are modelled on the heap. Hinrichsen et al. [2020, 2021a] developed the Actris mechanisation that this work is built on top of. Both lines of work focus on different properties than type safety.

Gay et al. [2020] explored various notions of duality, mechanising their results in Agda, and demonstrate that allowing duality to distribute over the recursive μ-operator yields an unsound system when type variables appear in messages, as the message type could change in tandem with the dualisation of the recursion, making endpoints disagree on the type of exchanged values. In our setup duality does not distribute over μ. Instead recursive definitions must be unfolded to expose the session type before duality can be applied, rendering the recursion and message type unchanged. Even so, we can drop down to Actris and use Löb induction to prove (subtyping) properties of recursive types and their duals.

Castro et al. [2020] focused on the metatheory of binary session types for synchronous communication, and prove in Coq, using the locally nameless approach to variable binding, subject reduction and that typing judgements are preserved by structural congruence.

Thiemann [2019] mechanised an intrinsically-typed definitional interpreter for a session-typed language with recursive types and subtyping in Agda. The mechanisation did, however, require a substantial amount of manual bookkeeping, in particular for properties about resource separation. Rouvoet et al. [2020] streamlined the intrinsically-typed approach by developing separation logic-like abstractions in Agda. They applied these abstractions to a small session-typed language without recursive types, subtyping, or polymorphism.

4.7 Conclusion

In this paper we demonstrate how the foundational semantic approach to type safety can be applied to session typing and how to construct and mechanise an extensible session-type system with support for manual typing proofs. The crux of the semantic approach is to use a program logic that is expressive enough to model all intended features (*e.g.,* channels, subtyping, polymorphism, recursion, locks/mutexes) while satisfying the required properties (*e.g.,* type safety). By building on top of the Iris and Actris frameworks we are able to inherit their constructs to mechanise such an extensible session-type system with little proof effort.

4.A The Complete Type System

This appendix includes an extensive overview of the mechanised semantic session-type system. Like the paper, all of the definitions and rules have been mechanised in the Coq proof assistant, and can be found in [Hinrichsen et al. 2021b].

In particular, the appendix shows the type and judgement definitions in Figure 4.7, the typing rules in Figures 4.8 and 4.9, and the subtyping rules in Figures 4.11 to 4.14.

As some of the details of the type system were omitted in the main text, we preface the overview with a cursory clarification of these. In particular, we introduce a streamlined approach for handling copyable versus uncopyable types, which allows unifying various typing rules (Section 4.A.1). We furthermore describe kinded subtyping and type equivalence (Section 4.A.2), shared reference types (Section 4.A.3), and discuss the internal versions of all judgements of the type system (Section 4.A.4).

We omit the typing rule for polymorphic sends from Section 4.3.5 because it can be derived from the original rule for send (Ty-ChanSend) along with the subsumption (Ty-Sub) and the subtyping for instantiating the binders of the send session type (SubTy-Send-In).

4.A.1 Uncopy

To handle copyable types, one typically has two rules for each construct that might move out ownership (one for non-copyable types and one for copyable types), *e.g.*:

$$\textsc{Ty-RefUniqLoad-Move}\quad \Gamma, x : \texttt{ref}_{\texttt{uniq}}\, A \vDash !\, x : A \dashv \Gamma, x : \texttt{ref}_{\texttt{uniq}}\, \texttt{any}$$

$$\frac{\texttt{copyable}\, A}{\Gamma, x : \texttt{ref}_{\texttt{uniq}}\, A \vDash !\, x : A \dashv \Gamma, x : \texttt{ref}_{\texttt{uniq}}\, A}\quad \textsc{Ty-RefUniqLoad-Copy}$$

The full version of our type system unifies these rules as a single rule Ty-RefUniqLoad using the `uncopy` type former. The `uncopy` type former acts as an inverse of the `copy` type former. When `uncopy` is applied to `copy` A, `copy` and `uncopy` cancel out, leaving the type A, as expressed by the subtyping rule SubTy-Uncopy-Elim. In combination with the rule SubTy-Copy-Intro, this means that the `uncopy` type former has no

effect on copyable types A, *i.e.,* if $\texttt{copyable}\, A$, then $\texttt{uncopy}\, A <: A$. However, when applied to a non-copyable type A, the `uncopy` type former has an effect, and thus cannot be stripped. This prevents the value from being used again, similarly to replacing the type by `any`.

The `uncopy` type former is defined in terms of the `coreP` modality of Iris (`coreP` itself is defined in terms of other logical primitives), which acts as a similar "inverse" to the persistence modality ($\square$). The definition and proof rules of the `coreP` modality can be found at https://gitlab.mpi-sws.org/iris/iris/-/blob/master/theories/bi/lib/core.v.

4.A.2 Kinded Subtyping and Type Equivalence

The subtyping relation $<:$ is kinded, *i.e.,* it takes arguments of type Type_k and its definition depends on the kind k. By making the subtyping relation kinded, we can unify subtyping rules that are identical for both type kinds, such as the rule Subty-Refl for reflexivity.

Additionally, to unify subtyping rules that go in both directions, such as the rule Subty-Rec-Unfold for unfolding recursive types, we define a relation for *type equivalence* $K <:> L$ as the symmetric closure of the subtyping relation:

$$K <:> L \triangleq K <: L \land L <: K$$

Similar to the subtyping relation, the relation for type equivalence is kinded so it applies to both term and session types.

4.A.3 Shared References

We also have an additional type former $\texttt{ref}_{\texttt{shr}}$, which is not mentioned in the main text of the paper. This is the type of *shared references*, or references that can be freely duplicated and shared between threads, but whose type is not allowed to change by writing new values. Moreover, shared references can only hold values of a copyable type, to prevent values from being copied by reading and writing to a reference.

The definition of the type former $\texttt{ref}_{\texttt{shr}}$ for shared references is standard in logical relation developments in Iris. It is defined in terms of Iris *invariants*, written $\boxed{P}$, which contain a proposition P. Invariants are always persistent (even if the proposition P itself is not), meaning they can be freely duplicated. Moreover, it is possible to *open* an invariant to gain access to the proposition P inside, as long as that is restricted to an atomic program step, and the invariant is re-established by reproving P at the end of the atomic step In practice, this means that it is only possible to apply atomic read and write operations to shared references, and the fact that invariants must be re-established ensures that we cannot *change* the type of the value contained in the reference, in contrast to the store rule for unique references $\texttt{ref}_{\texttt{uniq}}$.

4.A.4 Internal Judgements

In Section 4.5 we remarked that in the Coq mechanisation we defined the typing judgement as an internal definition in Iris, instead of as an external definition in the

meta logic. In the full version of the type system, we use the same treatment for the typing judgements. To make sure that the judgements behave like ordinary propositions of higher-order logic (instead of propositions that hold ownership), their definitions include the *plainly* modality (■). This modality carves out the step-indexed subset of the Iris logic. The rules of the plainly modality can be found in https://gitlab.mpi-sws.org/iris/iris/-/blob/master/theories/bi/plainly.v.

As a result of defining all judgements as internal notions, all typing rules are in fact implications in the Iris logic.

Term Types:

$$
\begin{aligned}
\mathsf{Type}_\bigstar &\triangleq \mathsf{Val} \to \mathsf{iProp} \\
\mathtt{any} &\triangleq \lambda w.\, \mathsf{True} \\
\mathbf{1} &\triangleq \lambda w.\, w \in \{()\} \\
\mathbf{B} &\triangleq \lambda w.\, w \in \mathbb{B} \\
\mathbf{Z} &\triangleq \lambda w.\, w \in \mathbb{Z} \\
\mathtt{ref}_{\mathtt{uniq}}\, A &\triangleq \lambda w.\, \exists v.\, w \in \mathsf{Loc} * (w \mapsto v) * \triangleright (A\, v) \\
\mathtt{ref}_{\mathtt{shr}}\, A &\triangleq \lambda w.\, (w \in \mathsf{Loc}) * \boxed{\exists v.\, (w \mapsto v) * \square (A\, v)} \\
A_1 \times A_2 &\triangleq \lambda w.\, \exists w_1, w_2.\, w = (w_1, w_2) \ * \triangleright (A_1\, w_1) * \triangleright (A_2\, w_2) \\
A_1 + A_2 &\triangleq \lambda w.\, \exists v.\, (w = \mathtt{inl}\ v * \triangleright (A_1\, v)) \ \lor (w = \mathtt{inr}\ v * \triangleright (A_2\, v)) \\
A \multimap B &\triangleq \lambda w.\, \forall v.\, \triangleright (A\, v) \mathrel{-\!\!*} \mathsf{wp}\ (w\ v)\ \{B\} \\
\mathtt{chan}\ S &\triangleq \lambda w.\, w \rightarrowtail S \\
\mathtt{copy}\, A &\triangleq \lambda w.\, \square (A\, w) \\
A \to B &\triangleq \mathtt{copy}\, (A \multimap B) \\
\mathtt{uncopy}\, A &\triangleq \lambda w.\, \mathtt{coreP}\ (A\, w) \\
\mu\, (X : k).\, K &\triangleq \mu (X : \mathsf{Type}_k).\, K \qquad (K \text{ is contractive in } X) \\
\forall (X : k).\, A &\triangleq \lambda w.\, \forall (X : \mathsf{Type}_k).\, \mathsf{wp}\ (w\ ())\ \{A\} \\
\exists (X : k).\, A &\triangleq \lambda w.\, \exists (X : \mathsf{Type}_k).\, \triangleright (A\, w) \\
\mathtt{mutex}\, A &\triangleq \lambda w.\, \exists lk, \ell.\, (w = (lk, \ell)) * \mathsf{is\_lock}\ lk\ (\exists v.\, (\ell \mapsto v) * \triangleright (A\, v)) \\
\overline{\mathtt{mutex}}\, A &\triangleq \lambda w.\, \exists lk, \ell.\, (w = (lk, \ell)) * \mathsf{is\_lock}\ lk\ (\exists v.\, (\ell \mapsto v) * \triangleright (A\, v)) * (\ell \mapsto -)
\end{aligned}
$$

Session Types:

$$
\begin{aligned}
\mathsf{Type}_\blacklozenge &\triangleq \mathsf{iProto} \\
\mathtt{end} &\triangleq \mathbf{end} \\
\mathbf{!} A.\, S &\triangleq \mathbf{!}\, (v : \mathsf{Val})\, \langle v \rangle \{A\, v\}.\, S \\
\mathbf{?} A.\, S &\triangleq \mathbf{?} (v : \mathsf{Val})\, \langle v \rangle \{A\, v\}.\, S \\
\mathbf{!}_{\vec{X}:\vec{k}}\, A.\, S &\triangleq \mathbf{!}\, (\vec{X} : \vec{\mathsf{Type}}_k)(v : \mathsf{Val})\, \langle v \rangle \{A\, v\}.\, S \\
\mathbf{?}_{\vec{X}:\vec{k}}\, A.\, S &\triangleq \mathbf{?} (\vec{X} : \vec{\mathsf{Type}}_k)(v : \mathsf{Val})\, \langle v \rangle \{A\, v\}.\, S \\
\oplus \{\vec{S}\} &\triangleq \mathbf{!}\, (l : \mathbb{Z})\, \langle l \rangle \left\{ l \in \mathsf{dom}(\vec{S}) \right\}.\, \vec{S}(l) \\
\& \{\vec{S}\} &\triangleq \mathbf{?} (l : \mathbb{Z})\, \langle l \rangle \left\{ l \in \mathsf{dom}(\vec{S}) \right\}.\, \vec{S}(l)
\end{aligned}
$$

Judgements:

$$
\begin{aligned}
\Gamma \vDash \sigma &\triangleq \ast_{(x,A) \in \Gamma}\, .\ A(\sigma(x)) \\
\Gamma \vDash e : A \dashv \Gamma' &\triangleq \blacksquare (\forall \sigma.\, (\Gamma \vDash \sigma) \mathrel{-\!\!*} \mathsf{wp}\ e[\sigma]\ \{v. A\, v * (\Gamma' \vDash \sigma)\}) \\
A <: B &\triangleq \blacksquare (\forall v.\, A\ v \mathrel{-\!\!*} B\ v) \\
S <: T &\triangleq \blacksquare (S \sqsubseteq T) \\
K <:> L &\triangleq K <: L \land L <: K \\
\Gamma <:_{\mathtt{ctx}} \Gamma' &\triangleq \blacksquare (\forall \sigma.\, (\Gamma \vDash \sigma) \mathrel{-\!\!*} (\Gamma' \vDash \sigma)) \\
\mathtt{copyable}\, A &\triangleq A <: \mathtt{copy}\, A
\end{aligned}
$$

Figure 4.7: Typing judgements and type formers.

Basics:

$$\textsc{Ty-Unit}\quad \Gamma \vDash () : \mathbf{1} \dashv \Gamma \qquad \textsc{Ty-Bool}\quad \Gamma \vDash b : \mathbf{B} \dashv \Gamma \qquad \textsc{Ty-Int}\quad \Gamma \vDash i : \mathbf{Z} \dashv \Gamma$$

$$\textsc{Ty-Neg}\ \frac{\Gamma \vDash e : \mathbf{B} \dashv \Gamma'}{\Gamma \vDash \neg e : \mathbf{B} \dashv \Gamma'}$$

$$\textsc{Ty-Arith}\ \frac{\Gamma \vDash e_2 : \mathbf{Z} \dashv \Gamma' \qquad \Gamma' \vDash e_1 : \mathbf{Z} \dashv \Gamma'' \qquad \mathsf{op} \in \{+, -\}}{\Gamma \vDash e_1\ \mathsf{op}\ e_2 : \mathbf{Z} \dashv \Gamma''}$$

$$\textsc{Ty-Cond}\ \frac{\Gamma \vDash e_2 : \mathbf{Z} \dashv \Gamma' \qquad \Gamma' \vDash e_1 : \mathbf{Z} \dashv \Gamma'' \qquad \mathsf{op} \in \{=, \leq\}}{\Gamma \vDash e_1\ \mathsf{op}\ e_2 : \mathbf{B} \dashv \Gamma''}$$

$$\textsc{Ty-If}\ \frac{\Gamma \vDash e_1 : \mathbb{B} \dashv \Gamma' \qquad \Gamma' \vDash e_2 : A \dashv \Gamma'' \qquad \Gamma' \vDash e_3 : A \dashv \Gamma''}{\Gamma \vDash \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 : A \dashv \Gamma''}$$

$$\textsc{Ty-Var}\ \Gamma, x : A \vDash x : A \dashv \Gamma, x : \mathsf{uncopy}\, A \qquad \textsc{Ty-Lam}\ \frac{\Gamma, x : A \vDash e : B \dashv \Gamma''}{\Gamma \cdot \Gamma' \vDash \lambda x.\, e : A \multimap B \dashv \Gamma'}$$

$$\textsc{Ty-Rec}\ \frac{\begin{array}{c}\Gamma = x_1 : A_1, \ldots, x_n : A_n \\ \Gamma_{\text{copy}} = x_1 : \mathsf{uncopy}\, A_1, \ldots, x_n : \mathsf{uncopy}\, A_n \\ \Gamma_{\text{copy}}, f : A \to B, x : A \vDash e : B \dashv \Gamma''\end{array}}{\Gamma \cdot \Gamma' \vDash \mathtt{rec}\ f\ x := e : A \to B \dashv \Gamma'}$$

$$\textsc{Ty-App}\ \frac{\Gamma \vDash e_2 : A \dashv \Gamma' \qquad \Gamma' \vDash e_1 : A \multimap B \dashv \Gamma''}{\Gamma \vDash e_1\ e_2 : B \dashv \Gamma''}$$

$$\textsc{Ty-Let}\ \frac{\Gamma_1 \vDash e_1 : A \dashv \Gamma_2 \qquad \Gamma_2, x : A \vDash e_2 : B \dashv \Gamma_3}{\Gamma_1 \vDash \mathtt{let}\ x := e_1\ \mathtt{in}\ e_2 : B \dashv \Gamma_3 \setminus x}$$

$$\textsc{Ty-Par}\ \frac{\Gamma_1 \vDash e_1 : A_1 \dashv \Gamma_1' \qquad \Gamma_2 \vDash e_2 : A_2 \dashv \Gamma_2'}{\Gamma_1 \cdot \Gamma_2 \vDash e_1 \mid\mid e_2 : A_1 \times A_2 \dashv \Gamma_1' \cdot \Gamma_2'}$$

$$\textsc{Ty-Sub}\ \frac{\Gamma_1 <:_{\mathbf{ctx}} \Gamma_1' \qquad \Gamma_1' \vDash e : A \dashv \Gamma_2' \qquad A <: B \qquad \Gamma_2' <:_{\mathbf{ctx}} \Gamma_2}{\Gamma_1 \vDash e : B \dashv \Gamma_2}$$

Figure 4.8: Term typing rules.

Product and Sums:

$$\textsc{Ty-Pair}\ \frac{\Gamma \vDash e_2 : A_2 \dashv \Gamma' \qquad \Gamma' \vDash e_1 : A_1 \dashv \Gamma''}{\Gamma \vDash (e_1, e_2) : A_1 \times A_2 \dashv \Gamma''} \qquad \textsc{Ty-InL}\ \frac{\Gamma \vDash e : A \dashv \Gamma'}{\Gamma \vDash \texttt{inl}\ e : A + B \dashv \Gamma'}$$

$$\textsc{Ty-InR}\ \frac{\Gamma \vDash e : B \dashv \Gamma'}{\Gamma \vDash \texttt{inr}\ e : A + B \dashv \Gamma'} \qquad \textsc{Ty-Fst}\ \ \Gamma, x : A_1 \times A_2 \vDash \texttt{fst}\ x : A_1 \dashv \Gamma, x : \texttt{uncopy}\, A_1 \times A_2$$

$$\textsc{Ty-Snd}\ \ \Gamma, x : A_1 \times A_2 \vDash \texttt{snd}\ x : A_2 \dashv \Gamma, x : A_1 \times \texttt{uncopy}\, A_2$$

$$\textsc{Ty-Case}\ \frac{\Gamma \vDash e_1 : A + B \dashv \Gamma' \qquad \Gamma' \vDash e_2 : A \multimap C \dashv \Gamma'' \qquad \Gamma' \vDash e_3 : B \multimap C \dashv \Gamma''}{\Gamma \vDash \texttt{case}\ e_1\ e_2\ e_3 : C \dashv \Gamma''}$$

Polymorphism:

$$\textsc{Ty-TLam}\ \frac{\Gamma \vDash e : A \dashv \Gamma'' \qquad X \notin FV(\Gamma, \Gamma')}{\Gamma \cdot \Gamma' \vDash \lambda\_.\, e : \forall X.\, A \dashv \Gamma'} \qquad \textsc{Ty-TApp}\ \frac{\Gamma \vDash e : \forall X.\, A \dashv \Gamma'}{\Gamma \vDash e\ () : A[K/X] \dashv \Gamma'} \qquad \textsc{Ty-Pack}\ \frac{\Gamma \vDash e : A[K/X] \dashv \Gamma'}{\Gamma \vDash e : \exists X.\, A \dashv \Gamma'}$$

$$\textsc{Ty-Unpack}\ \frac{\Gamma \vDash e_1 : \exists X.\, A \dashv \Gamma' \qquad \Gamma', x : A \vDash e_2 : B \dashv \Gamma'' \qquad X \notin FV(\Gamma, \Gamma'', B)}{\Gamma \vDash \texttt{let}\ x := e_1\ \texttt{in}\ e_2 : B \dashv \Gamma'' \setminus x}$$

References:

$$\textsc{Ty-ToRefShr}\ \frac{\Gamma \vDash e : \texttt{ref}_{\texttt{uniq}}\,(\texttt{copy}\, A) \dashv \Gamma'}{\Gamma \vDash e : \texttt{ref}_{\texttt{shr}}\ A \dashv \Gamma'} \qquad \textsc{Ty-RefShrLoad}\ \frac{\Gamma \vDash e : \texttt{ref}_{\texttt{shr}}\ A \dashv \Gamma'}{\Gamma \vDash\, !\, e : A \dashv \Gamma'}$$

$$\textsc{Ty-RefShrStore}\ \frac{\Gamma \vDash e_2 : \texttt{copy}\, A \dashv \Gamma' \qquad \Gamma' \vDash e_1 : \texttt{ref}_{\texttt{shr}}\ A \dashv \Gamma''}{\Gamma \vDash e_1\ \leftarrow e_2 : \mathbf{1} \dashv \Gamma''} \qquad \textsc{Ty-RefUniqAlloc}\ \frac{\Gamma \vDash e : A \dashv \Gamma'}{\Gamma \vDash \texttt{ref}\ e : \texttt{ref}_{\texttt{uniq}}\, A \dashv \Gamma'}$$

$$\textsc{Ty-RefUniqFree}\ \frac{\Gamma \vDash e : \texttt{ref}_{\texttt{uniq}}\, A \dashv \Gamma'}{\Gamma \vDash \texttt{free}\, e : \mathbf{1} \dashv \Gamma'} \qquad \textsc{Ty-RefUniqStore}\ \frac{\Gamma \vDash e : B \dashv \Gamma', x : \texttt{ref}_{\texttt{uniq}}\, A}{\Gamma \vDash x\ \leftarrow e : \mathbf{1} \dashv \Gamma', x : \texttt{ref}_{\texttt{uniq}}\, B}$$

$$\textsc{Ty-RefUniqLoad}\ \ \Gamma, x : \texttt{ref}_{\texttt{uniq}}\, A \vDash\, !\, x : A \dashv \Gamma, x : \texttt{ref}_{\texttt{uniq}}\,(\texttt{uncopy}\, A)$$

Figure 4.9: Term typing rules (cont.)

Channels:

$$\textsc{Ty-ChanAlloc}\quad \Gamma \vDash \texttt{new\_chan} : \mathbf{1} \to \texttt{chan}\ S \times \texttt{chan}\ \overline{S} \dashv \Gamma$$

$$\textsc{Ty-ChanSend}\quad \frac{\Gamma \vDash e : A \dashv \Gamma', x : \texttt{chan}\ (!A.\, S)}{\Gamma \vDash \texttt{send}\ x\ e : \mathbf{1} \dashv \Gamma', x : \texttt{chan}\ S}$$

$$\textsc{Ty-ChanRecv}\quad \Gamma, x : \texttt{chan}\ (?A.\, S) \vDash \texttt{recv}\ x : A \dashv \Gamma, x : \texttt{chan}\ S$$

$$\textsc{Ty-ChanRecvPoly}\quad \frac{\Gamma, x : \texttt{chan}\ S, y : A \vDash e : B \dashv \Gamma' \qquad \vec{X} \notin FV(\Gamma, \Gamma', B)}{\Gamma, x : \texttt{chan}\ (?_{\vec{X}:\vec{k}}\, A.\, S) \vDash \texttt{let}\ y := \texttt{recv}\ x\ \texttt{in}\ e : B \dashv \Gamma' \setminus \{y\}}$$

$$\textsc{Ty-Select}\quad \frac{1 \le i \le n}{\Gamma, x : \texttt{chan}\ (\oplus\{l_1 : S_1, \ldots, l_n : S_n\}) \vDash \texttt{select}\ x\ l_i : \mathbf{1} \dashv \Gamma, x : \texttt{chan}\ S_i}$$

$$\textsc{Ty-Branch}\quad \frac{\Gamma, x : \texttt{chan}\ S_1 \vDash e_1 : A \dashv \Gamma' \quad \cdots \quad \Gamma, x : \texttt{chan}\ S_n \vDash e_n : A \dashv \Gamma'}{\Gamma, x : \texttt{chan}\ (\&\{l_1 : S_1, \ldots, l_n : S_n\}) \vDash \texttt{branch}\ x\ \texttt{with}\ l_1 \Rightarrow e_1 \mid \ldots \mid l_n \Rightarrow e_n : A \dashv \Gamma'}$$

Locks:

$$\textsc{Ty-MutexAlloc}\quad \Gamma \vDash \mathsf{newmutex} : A \to \texttt{mutex}\ A \dashv \Gamma$$

$$\textsc{Ty-MutexAcquire}\quad \Gamma, x : \texttt{mutex}\ A \vDash \mathsf{acquiremutex}\ x : A \dashv \Gamma, x : \overline{\texttt{mutex}}\ A$$

$$\textsc{Ty-MutexRelease}\quad \frac{\Gamma \vDash e : A \dashv \Gamma', x : \overline{\texttt{mutex}}\ A}{\Gamma \vDash \mathsf{releasemutex}\ x\ e : \mathbf{1} \dashv \Gamma', x : \texttt{mutex}\ A}$$

Figure 4.10: Term typing rules (cont.)

Subtyping Properties:

SubTy-Refl
$$K <: K$$

SubTy-Trans
$$\frac{K <: L \qquad L <: M}{K <: M}$$

SubTy-Bi
$$\frac{K <: L \qquad L <: K}{K <:> L}$$

SubTy-Bi-Refl
$$K <:> K$$

SubTy-Bi-Trans
$$\frac{K <:> L \qquad L <:> M}{K <:> M}$$

SubTy-Bi-Trans-Left
$$\frac{K <:> L \qquad L <: M}{K <: M}$$

SubTy-Bi-Trans-Right
$$\frac{K <: L \qquad L <:> M}{K <: M}$$

SubTy-Bi-Sym
$$\frac{L <:> K}{K <:> L}$$

SubTy-Rec-Unfold
$$\mu X.\, K <:> K(\mu X.\, K)$$

Term Subtyping:

SubTy-Any
$$A <: \texttt{any}$$

SubTy-Lolli
$$\frac{C <: A \qquad B <: D}{A \multimap B <: C \multimap D}$$

SubTy-Arr
$$\frac{C <: A \qquad B <: D}{A \to B <: C \to D}$$

SubTy-Product
$$\frac{A <: C \qquad B <: D}{A \times B <: C \times D}$$

SubTy-Sum
$$\frac{A <: C \qquad B <: D}{A + B <: C + D}$$

SubTy-Forall
$$\frac{\forall X.\, (A <: B)}{\forall X.\, A <: \forall X.\, B}$$

SubTy-Exist
$$\frac{\forall X.\, (A <: B)}{\exists X.\, A <: \exists X.\, B}$$

SubTy-Exist-Elim
$$A[K/X] <: \exists X.\, A$$

SubTy-Ref-Uniq
$$\frac{A <: B}{\texttt{ref}_{\texttt{uniq}}\, A <: \texttt{ref}_{\texttt{uniq}}\, B}$$

SubTy-Ref-Shr
$$\frac{A <:> B}{\texttt{ref}_{\texttt{shr}}\, A <: \texttt{ref}_{\texttt{shr}}\, B}$$

SubTy-Chan
$$\frac{S <: T}{\texttt{chan}\, S <: \texttt{chan}\, T}$$

SubTy-Mutex
$$\frac{A <:> B}{\texttt{mutex}\, A <: \texttt{mutex}\, B}$$

SubTy-MutexGuard
$$\frac{A <:> B}{\overline{\texttt{mutex}}\, A <: \overline{\texttt{mutex}}\, B}$$

Figure 4.11: Subtyping rules.

Copyable Types:

$$\textsc{SubTy-Copy}\ \frac{A <: B}{\texttt{copy}\,A <: \texttt{copy}\,B} \qquad \textsc{SubTy-Copy-Intro}\ \frac{\texttt{copyable}\,A}{A <: \texttt{copy}\,A} \qquad \textsc{SubTy-Copy-Elim}\ \ \texttt{copy}\,A <: A$$

$$\textsc{SubTy-Uncopy}\ \frac{A <: B}{\texttt{uncopy}\,A <: \texttt{uncopy}\,B} \qquad \textsc{SubTy-Uncopy-Intro}\ \ A <: \texttt{uncopy}\,A \qquad \textsc{SubTy-Uncopy-Elim}\ \ \texttt{uncopy}\,(\texttt{copy}\,A) <: A$$

$$\textsc{SubTy-Copyable-Copy}\ \ \texttt{copyable}\,(\texttt{copy}\,A) \qquad \textsc{SubTy-Copyable-Uncopy}\ \ \texttt{copyable}\,(\texttt{uncopy}\,A) \qquad \textsc{SubTy-Copyable-Any}\ \ \texttt{copyable any}$$

$$\textsc{SubTy-Copyable-Unit}\ \ \texttt{copyable}\,\mathbf{1} \qquad \textsc{SubTy-Copyable-Bool}\ \ \texttt{copyable}\,\mathbf{B} \qquad \textsc{SubTy-Copyable-Int}\ \ \texttt{copyable}\,\mathbf{Z}$$

$$\textsc{SubTy-Copyable-Product}\ \frac{\texttt{copyable}\,A \qquad \texttt{copyable}\,B}{\texttt{copyable}\,(A \times B)} \qquad \textsc{SubTy-Copyable-Sum}\ \frac{\texttt{copyable}\,A \qquad \texttt{copyable}\,B}{\texttt{copyable}\,(A + B)}$$

$$\textsc{SubTy-Copyable-Exists}\ \frac{\forall X.\,\texttt{copyable}\,A}{\texttt{copyable}\,(\exists X.\,A)} \qquad \textsc{SubTy-Copyable-RefShr}\ \ \texttt{copyable}\,(\texttt{ref}_{\texttt{shr}}\ X)$$

$$\textsc{SubTy-Copyable-Mutex}\ \ \texttt{copyable}\,(\texttt{mutex}\,X)$$

Context Subtyping:

$$\textsc{Ctx-Permute}\ \frac{\Gamma' \text{ is a permutation of } \Gamma}{\Gamma <:_{\mathbf{ctx}} \Gamma'} \qquad \textsc{Ctx-Refl}\ \ \Gamma <:_{\mathbf{ctx}} \Gamma \qquad \textsc{Ctx-Trans}\ \frac{\Gamma_1 <:_{\mathbf{ctx}} \Gamma_2 \qquad \Gamma_2 <:_{\mathbf{ctx}} \Gamma_3}{\Gamma_1 <:_{\mathbf{ctx}} \Gamma_3}$$

$$\textsc{Ctx-Nil}\ \ \Gamma <:_{\mathbf{ctx}} [\,] \qquad \textsc{Ctx-Cons}\ \frac{A <: B \qquad \Gamma <:_{\mathbf{ctx}} \Gamma'}{x : A, \Gamma <:_{\mathbf{ctx}} x : B, \Gamma'} \qquad \textsc{Ctx-App}\ \frac{\Gamma_1 <:_{\mathbf{ctx}} \Gamma_2 \qquad \Gamma_1' <:_{\mathbf{ctx}} \Gamma_2'}{\Gamma_1 \cdot \Gamma_1' <:_{\mathbf{ctx}} \Gamma_2 \cdot \Gamma_2'}$$

$$\textsc{Ctx-Copy}\ \ x : A <:_{\mathbf{ctx}} x : A, x : \texttt{uncopy}\,A \qquad \textsc{Ctx-Copyable}\ \frac{\texttt{copyable}\,A}{x : A <:_{\mathbf{ctx}} x : A, x : A}$$

Figure 4.12: Subtyping rules (cont.)

Session Subtyping:

$$\textsc{SubTy-Send}\ \frac{B <: A \qquad S <: T}{!A.\,S <: !B.\,T} \qquad \textsc{SubTy-Recv}\ \frac{A <: B \qquad S <: T}{?A.\,S <: ?B.\,T} \qquad \textsc{SubTy-Send-In}\ !_{(\vec{X}:\vec{k})}\,A.\,S <: !A[\vec{K}/\vec{X}].\,S[\vec{K}/\vec{X}]$$

$$\textsc{SubTy-Recv-In}\ ?A[\vec{K}/\vec{X}].\,S[\vec{K}/\vec{X}] <: ?_{(\vec{X}:\vec{k})}\,A.\,S \qquad \textsc{SubTy-Send-Out}\ \frac{S <: !A.\,T}{S <: !_{(\vec{X}:\vec{k})}\,A.\,T} \qquad \textsc{SubTy-Recv-Out}\ \frac{?A.\,S <: T}{?_{(\vec{X}:\vec{k})}\,A.\,S <: T}$$

$$\textsc{SubTy-Select}\ \frac{\forall i.\ \vec{S_i} <: \vec{T_i}}{\oplus\{\vec{l_i} : \vec{S_i}\}_{i\in\vec{i}} <: \oplus\{\vec{l_i} : \vec{T_i}\}_{i\in\vec{i}}} \qquad \textsc{SubTy-Select-SubsetEq}\ \frac{\vec{j} \subseteq \vec{i}}{\oplus\{\vec{l_i} : \vec{S_i}\}_{i\in\vec{i}} <: \oplus\{\vec{l_j} : \vec{S_j}\}_{j\in\vec{j}}}$$

$$\textsc{SubTy-Branch}\ \frac{\forall i.\ \vec{S_i} <: \vec{T_i}}{\&\{\vec{l_i} : \vec{S_i}\}_{i\in\vec{i}} <: \&\{\vec{l_i} : \vec{T_i}\}_{i\in\vec{i}}} \qquad \textsc{SubTy-Branch-SubsetEq}\ \frac{\vec{i} \subseteq \vec{j}}{\&\{\vec{l_i} : \vec{S_i}\}_{i\in\vec{i}} <: \&\{\vec{l_j} : \vec{S_j}\}_{j\in\vec{j}}}$$

$$\textsc{SubTy-Swap-Recv-Send}\ ?A.\,!B.\,S <: !B.\,?A.\,S$$

$$\textsc{SubTy-Swap-Branch-Send}\ \&\{l_1 : !A.\,S_1, \ldots, l_n : !A.\,S_n\} <: !A.\,\&\{l_1 : S_1, \ldots, l_n : S_n\}$$

$$\textsc{SubTy-Swap-Recv-Select}\ ?A.\ \oplus\{l_1 : S_1, \ldots, l_n : S_n\} <: \oplus\{l_1 : ?A.\,S_1, \ldots, l_n : ?A.\,S_n\}$$

$$\begin{aligned}\textsc{SubTy-Swap-Branch-Select}\ &\&\{l_1 : \oplus\{l'_1 : S_{(1,1)}, \ldots, l'_m : S_{(1,m)}\}, \ \ldots, \\ &l_n : \oplus\{l'_1 : S_{(n,1)}, \ldots, l'_m : S_{(n,m)}\}\} <: \oplus\{l'_1 : \&\{l_1 : S_{(1,1)}, \ldots, l_n : S_{(n,1)}\}, \ \ldots, \\ &l'_m : \&\{l_1 : S_{(n,1)}, \ldots, l_n : S_{(n,m)}\}\}\end{aligned}$$

Figure 4.13: Subtyping rules (cont.)

Duality Subtyping:

$$\textsc{SubTy-Dual}\ \frac{T <: S}{\overline{S} <: \overline{T}} \qquad \textsc{SubTy-Dual-Left}\ \frac{\overline{T} <: S}{\overline{S} <: T} \qquad \textsc{SubTy-Dual-Right}\ \frac{T <: \overline{S}}{S <: \overline{T}}$$

$$\textsc{SubTy-Dual-Send}\ \overline{!_{\vec{X}}\, A.\, S} <:> ?_{\vec{X}}\, A.\, \overline{S} \qquad \textsc{SubTy-Dual-Recv}\ \overline{?_{\vec{X}}\, A.\, S} <:> !_{\vec{X}}\, A.\, \overline{S} \qquad \textsc{SubTy-Dual-End}\ \overline{\texttt{end}} <:> \texttt{end}$$

$$\textsc{SubTy-Dual-Select}\ \overline{\oplus\{l_1 : S_1, \ldots, l_n : S_n\}} <:> \&\{l_1 : \overline{S_1}, \ldots, l_n : \overline{S_n}\}$$

$$\textsc{SubTy-Dual-Branch}\ \overline{\&\{l_1 : S_1, \ldots, l_n : S_n\}} <:> \oplus\{l_1 : \overline{S_1}, \ldots, l_n : \overline{S_n}\}$$

Append Subtyping:

$$\textsc{SubTy-App}\ \frac{S <: U \qquad T <: V}{S \cdot T <: U \cdot V} \qquad \textsc{SubTy-App-Assoc}\ S \cdot (T \cdot U) <:> (S \cdot T) \cdot U$$

$$\textsc{SubTy-App-Send}\ (!_{\vec{X}}\, A.\, S) \cdot T <:> !_{\vec{X}}\, A.\, (S \cdot T) \qquad \textsc{SubTy-App-Recv}\ (?_{\vec{X}}\, A.\, S) \cdot T <:> ?_{\vec{X}}\, A.\, (S \cdot T)$$

$$\textsc{SubTy-App-Select}\ (\oplus\{l_1 : S_1, \ldots, l_n : S_n\}) \cdot T <:> \oplus\{l_1 : S_1 \cdot T, \ldots, l_n : S_n \cdot T\}$$

$$\textsc{SubTy-App-Branch}\ (\&\{l_1 : S_1, \ldots, l_n : S_n\}) \cdot T <:> \&\{l_1 : S_1 \cdot T, \ldots, l_n : S_n \cdot T\}$$

$$\textsc{SubTy-App-End-Right}\ S \cdot \texttt{end} <:> S \qquad \textsc{SubTy-App-End-Left}\ \texttt{end} \cdot S <:> S$$

Figure 4.14: Subtyping rules (cont.)

Bibliography

Amal Ahmed. 2004. *Semantics of types for mutable state*. Ph.D. Dissertation. Princeton University. https://dl.acm.org/doi/10.5555/1037736

Amal Ahmed, Andrew W. Appel, Christopher D. Richards, Kedar N. Swadi, Gang Tan, and Daniel C. Wang. 2010. Semantic foundations for typed assembly languages. *TOPLAS* 32, 3 (2010). https://doi.org/10.1145/1709093.1709094

Akka. 2021. The Akka Project. (2021). https://akka.io/

Akka.NET. 2021. The Akka.NET Project. (2021). https://getakka.net/

Pierre America and Jan J. M. M. Rutten. 1989. Solving Reflexive Domain Equations in a Category of Complete Metric Spaces. *JCSS* 39, 3 (1989). https://doi.org/10.1016/0022-0000(89)90027-5

Andrew W. Appel. 2011. Verified Software Toolchain. In *ESOP (LNCS, Vol. 7226)*. https://doi.org/10.1007/978-3-642-19718-5_1

Andrew W. Appel. 2014. *Program Logics - for Certified Compilers*. Cambridge University Press. http://www.cambridge.org/de/academic/subjects/computer-science/programming-languages-and-applied-logic/program-logics-certified-compilers?format=HB

Andrew W. Appel and David A. McAllester. 2001. An indexed model of recursive types for foundational proof-carrying code. *TOPLAS* 23, 5 (2001). https://doi.org/10.1145/504709.504712

Andrew W. Appel, Paul-André Melliès, Christopher D. Richards, and Jérôme Vouillon. 2007. A very modal model of a modern, major, general type system. In *POPL*. https://doi.org/10.1145/1190216.1190235

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*. https://doi.org/10.1007/978-3-319-30936-1_2

Mehdi Bagherzadeh, Nicholas Fireman, Anas Shawesh, and Raffi Khatchadourian. 2020. Actor concurrency bugs: a comprehensive study on symptoms, root causes, API usages, and differences. *PACMPL* 4, OOPSLA (2020). https://doi.org/10.1145/3428282

Stephanie Balzer and Frank Pfenning. 2017. Manifest Sharing with Session Types. *PACMPL* 1, ICFP (2017). https://doi.org/10.1145/3110281

Stephanie Balzer, Bernardo Toninho, and Frank Pfenning. 2019. Manifest Deadlock-Freedom for Shared Session Types. In *ESOP (LNCS, Vol. 11423)*. https://doi.org/10.1007/978-3-030-17184-1_22

Bedrock Systems A/S. 2021. The Bedrock Systems Project. (2021). https://bedrocksystems.com/

Lennart Beringer, Adam Petcher, Katherine Q. Ye, and Andrew W. Appel. 2015. Verified Correctness and Security of OpenSSL HMAC. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/beringer

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

Lars Birkedal, Rasmus Ejlers Møgelberg, Jan Schwinghammer, and Kristian Støvring. 2012. First steps in synthetic guarded domain theory: step-indexing in the topos of trees. *LMCS* 8, 4 (2012). https://doi.org/10.2168/LMCS-8(4:1)2012

Lars Birkedal, Kristian Støvring, and Jacob Thamsborg. 2010. The category-theoretic solution of recursive metric-space equations. *TCS* 411, 47 (2010). https://doi.org/10.1016/j.tcs.2010.07.010

Ales Bizjak, Daniel Gratzer, Robbert Krebbers, and Lars Birkedal. 2019. Iron: managing obligations in higher-order concurrent separation logic. *PACMPL* 3, POPL (2019). https://doi.org/10.1145/3290378

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

Michael Brandt and Fritz Henglein. 1998. Coinductive Axiomatization of Recursive Type Equality and Subtyping. *Fundamenta Informaticae* 33, 4 (1998). https://doi.org/10.3233/FI-1998-33401

Mario Bravetti, Marco Carbone, Julien Lange, Nobuko Yoshida, and Gianluigi Zavattaro. 2021. A Sound Algorithm for Asynchronous Session Subtyping and its Implementation. *LMCS* 17, 1 (2021). https://lmcs.episciences.org/7238

Mario Bravetti, Marco Carbone, and Gianluigi Zavattaro. 2017. Undecidability of asynchronous session subtyping. *Information and Computation* 256 (2017). https://doi.org/10.1016/j.ic.2017.07.010

Stephen D. Brookes. 2004. A Semantics for Concurrent Separation Logic. In *CONCUR (LNCS, Vol. 3170)*. https://doi.org/10.1007/978-3-540-28644-8_2

Luís Caires, Jorge A. Pérez, Frank Pfenning, and Bernardo Toninho. 2013. Behavioral Polymorphism and Parametricity in Session-Based Communication. In *ESOP (LNCS, Vol. 7792)*. https://doi.org/10.1007/978-3-642-37036-6_19

Luís Caires and Frank Pfenning. 2010. Session Types as Intuitionistic Linear Propositions. In *CONCUR (LNCS, Vol. 6269)*. https://doi.org/10.1007/978-3-642-15375-4_16

Marco Carbone, Fabrizio Montesi, Carsten Schürmann, and Nobuko Yoshida. 2017. Multiparty session types as coherence proofs. *Acta Informatica* 54, 3 (2017). https://doi.org/10.1007/s00236-016-0285-y

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

Ernie Cohen, Eyad Alkassar, Vladimir Boyarinov, Markus Dahlweid, Ulan Degenbaev, Mark A. Hillebrand, Bruno Langenstein, Dirk Leinenbach, Michal Moskal, Steven Obua, Wolfgang J. Paul, Hristo Pentchev, Elena Petrova, Thomas Santen, Norbert Schirmer, Sabine Schmaltz, Wolfram Schulte, Andrey Shadrin, Stephan Tobies, Alexandra Tsyban, and Sergey Tverdyshev. 2009. Invariants, Modularity, and Rights. In *PSI*. https://doi.org/10.1007/978-3-642-11486-1_4

Coq Development Team. 2021. The Coq Proof Assistant. (2021). https://coq.inria.fr

Andreea Costea, Wei-Ngan Chin, Shengchao Qin, and Florin Craciun. 2018. Automated Modular Verification for Relaxed Communication Protocols. In *APLAS*. https://doi.org/10.1007/978-3-030-02768-1_16

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

Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. 2014. TaDA: A Logic for Time and Data Abstraction. In *ECOOP*. https://doi.org/10.1007/978-3-662-44202-9_9

Ornela Dardha and Simon J. Gay. 2018. A New Linear Logic for Deadlock-Free Session-Typed Processes. In *FOSSACS (LNCS, Vol. 10803)*. https://doi.org/10.1007/978-3-319-89366-2_5

Ornela Dardha, Elena Giachino, and Davide Sangiorgi. 2012. Session Types Revisited. In *PPDP*. https://doi.org/10.1007/978-3-030-17184-1_22

Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 2015. *The Lean Theorem Prover (System Description)*. Cham. https://doi.org/10.1007/978-3-319-21401-6_26

Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*. http://www.usenix.org/events/osdi04/tech/dean.html

Derek Dreyer, Amal Ahmed, and Lars Birkedal. 2009. Logical Step-Indexed Logical Relations. In *LICS*. https://doi.org/10.1109/LICS.2009.34

Derek Dreyer, Georg Neis, Andreas Rossberg, and Lars Birkedal. 2010. A relational modal logic for higher-order stateful ADTs. In *POPL*. https://doi.org/10.1145/1706299.1706323

Derek Dreyer, Amin Timany, Robbert Krebbers, Lars Birkedal, and Ralf Jung. 2019. What Type Soundness Theorem Do You Really Want to Prove? SIGPLAN blog post, available at https://blog.sigplan.org/2019/10/17/what-type-soundness-theorem-do-you-really-want-to-prove/.

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

Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2018. ReLoC: A Mechanised Relational Logic for Fine-Grained Concurrency. In *LICS*. https://doi.org/10.1145/3209108.3209174

Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2020. Compositional Non-Interference for Fine-Grained Concurrent Programs. To appear in S&P'21.

Simon J. Gay. 2008. Bounded polymorphism in session types. *MSCS* 18, 5 (2008). https://doi.org/10.1017/S0960129508006944

Simon J. Gay. 2016. Subtyping Supports Safe Session Substitution. In *Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*. https://doi.org/10.1007/978-3-319-30936-1_5

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

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

Paolo G. Giarrusso, Léo Stefanesco, Amin Timany, Lars Birkedal, and Robbert Krebbers. 2020. Scala step-by-step: soundness for DOT with step-indexed logical relations in Iris. *PACMPL* 4, ICFP (2020). https://doi.org/10.1145/3408996

Liang Gu, Alexander Vaynberg, Bryan Ford, Zhong Shao, and David Costanzo. 2011. CertiKOS: a certified kernel for secure cloud computing. In *APSys '11 Asia Pacific Workshop on Systems, Shanghai, China, July 11-12, 2011*. https://doi.org/10.1145/2103799.2103803

Jafar Hamin and Bart Jacobs. 2019. Transferring Obligations Through Synchronizations. In *ECOOP*. https://doi.org/10.4230/LIPIcs.ECOOP.2019.19

Carl Hewitt, Peter Boehler Bishop, and Richard Steiger. 1973. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *IJCAI*. http://ijcai.org/Proceedings/73/Papers/027B.pdf

Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. 2020. Actris: Session-type based reasoning in separation logic. *PACMPL* 4, POPL (2020). https://doi.org/10.1145/3371074

Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. 2021a. Actris 2.0: Asynchronous session-type based reasoning in separation logic. (2021). Manuscript under review.

Jonas Kastberg Hinrichsen, Daniël Louwrink, Robbert Krebbers, and Jesper Bengtson. 2021b. Coq Mechanization of "Machine-checked semantic session typing". Archived version at https://zenodo.org/record/4322752, latest version at https://gitlab.mpi-sws.org/iris/actris.

Jonas Kastberg Hinrichsen, Daniël Louwrink, Robbert Krebbers, and Jesper Bengtson. 2021c. Machine-checked semantic session typing. In *CPP*. https://doi.org/10.1145/3437992.3439914

C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (1969). https://doi.org/10.1145/363235.363259

Aquinas Hobor, Andrew Appel, and Francesco Nardelli. 2008. Oracle Semantics for Concurrent Separation Logic. In *Programming Languages and Systems*. Lecture Notes in Computer Science, Vol. 4960. Springer Berlin / Heidelberg. http://dx.doi.org/10.1007/978-3-540-78739-6_27 10.1007/978-3-540-78739-6_27.

Kohei Honda. 1993. Types for Dyadic Interaction. In *CONCUR*. https://doi.org/10.1007/3-540-57208-2_35

Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. 1998. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *ESOP (LNCS, Vol. 1381)*. https://doi.org/10.1007/BFb0053567

Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty asynchronous session types. In *POPL*. https://doi.org/10.1145/1328438.1328472

Raymond Hu, Dimitrios Kouzapas, Olivier Pernet, Nobuko Yoshida, and Kohei Honda. 2010. Type-Safe Eventful Sessions in Java. In *ECOOP*. https://doi.org/10.1007/978-3-642-14107-2_16

Keigo Imai, Nobuko Yoshida, and Shoji Yuen. 2019. Session-OCaml: A session-based library with polarities and lenses. *Science of Computer Programming* 172 (2019). https://doi.org/10.1016/j.scico.2018.08.005

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

Samin S. Ishtiaq and Peter W. O'Hearn. 2001. BI as an Assertion Language for Mutable Data Structures. In *POPL*. https://dl.acm.org/doi/10.1145/373243.375719

Jacques-Henri Jourdan and Robbert Krebbers. 2018. Iris Tutorial at POPL. Available online at https://gitlab.mpi-sws.org/iris/tutorial-popl18.

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018a. RustBelt: Securing the Foundations of the Rust Programming Language. *PACMPL* 2, POPL (2018). https://doi.org/10.1145/3158154

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2021. Safe systems programming in Rust: The promise and the challenge. To appear in CACM.

Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Higher-Order Ghost State. In *ICFP*. https://doi.org/10.1145/2951913

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). https://doi.org/10.1017/S0956796818000151

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *POPL*. https://doi.org/10.1145/2676726.2676980

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: formal verification of an OS kernel. In *SOSP*. https://doi.org/10.1145/1629575.1629596

Naoki Kobayashi. 2006. A New Type System for Deadlock-Free Processes. In *CONCUR (LNCS, Vol. 4137)*. https://doi.org/10.1007/11817949_16

Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. 1996. Linearity and the Pi-Calculus. In *POPL*. https://doi.org/10.1145/237721.237804

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). https://doi.org/10.1145/3236772

Robbert Krebbers, Ralf Jung, Ales Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017a. The Essence of Higher-Order Concurrent Separation Logic. In *ESOP (LNCS, Vol. 10201)*. https://doi.org/10.1007/978-3-662-54434-1_26

Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017b. Interactive Proofs in Higher-Order Concurrent Separation Logic. In *POPL*. https://doi.org/10.1145/3093333.3009855

Neelakantan R. Krishnaswami, Aaron Turon, Derek Dreyer, and Deepak Garg. 2012. Superficially substructural types. In *ICFP*. https://doi.org/10.1145/2364527.2364536

Morten Krogh-Jespersen, Kasper Svendsen, and Lars Birkedal. 2017. A relational model of types-and-effects in higher-order concurrent separation logic. In *POPL*. https://doi.org/10.1145/3093333.3009877

Morten Krogh-Jespersen, Amin Timany, Marit Edna Ohlenbusch, Simon Oddershede Gregersen, and Lars Birkedal. 2020. Aneris: A Mechanised Logic for Modular Reasoning about Distributed Systems. In *ESOP*. https://doi.org/10.1007/978-3-030-44914-8_13

Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: a verified implementation of ML. In *POPL*. https://doi.org/10.1145/2535838.2535841

Leslie Lamport. 1977. Proving the Correctness of Multiprocess Programs. *IEEE TSE* 3, 2 (1977). https://doi.org/10.1109/TSE.1977.229904

Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. 2018. A static verification framework for message passing in Go using behavioural types. In *ICSE*. https://doi.org/10.1145/3180155.3180157

Xavier Leroy. 2006. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *POPL*. https://doi.org/10.1145/1111037.1111042

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

William Mansky, Andrew W. Appel, and Aleksey Nogin. 2017. A verified messaging system. *PACMPL* 1, OOPSLA (2017). https://doi.org/10.1145/3133911

Claude Marché, Christine Paulin-Mohring, and Xavier Urbain. 2004. The KRAKATOA tool for certificationof JAVA/JAVACARD programs annotated in JML. *JLAMP* 58, 1-2 (2004). https://doi.org/10.1016/j.jlap.2003.07.006

Robin Milner. 1978. A Theory of Type Polymorphism in Programming. *JCSS* 17, 3 (1978). https://doi.org/10.1016/0022-0000(78)90014-4

Dimitris Mostrous and Vasco Thudichum Vasconcelos. 2014. Affine Sessions. In *COORDINATION*. https://doi.org/10.1007/978-3-662-43376-8_8

Dimitris Mostrous and Nobuko Yoshida. 2015. Session typing and asynchronous subtyping for the higher-order π-calculus. *Information and Computation* 241 (2015). https://doi.org/10.1016/j.ic.2015.02.002

Dimitris Mostrous, Nobuko Yoshida, and Kohei Honda. 2009. Global Principal Typing in Partially Commutative Asynchronous Sessions. In *ESOP (LNCS, Vol. 5502)*. https://doi.org/10.1007/978-3-642-00590-9_23

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

Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. 2014. Communicating State Transition Systems for Fine-Grained Concurrent Resources. In *ESOP*. https://doi.org/10.1007/978-3-642-54833-8_16

Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. 2002. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. LNCS, Vol. 2283. https://doi.org/10.1007/3-540-45949-9

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

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*. https://doi.org/10.1007/978-3-642-24690-6_24

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

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

Luca Padovani. 2017. A simple library implementation of binary sessions. *JFP* 27 (2017). https://doi.org/10.1017/S0956796816000289

Jorge A. Pérez, Luís Caires, Frank Pfenning, and Bernardo Toninho. 2012. Linear Logical Relations for Session-Based Concurrency. In *ESOP (LNCS, Vol. 7211)*. https://doi.org/10.1007/978-3-642-28869-2_27

Jorge A. Pérez, Luís Caires, Frank Pfenning, and Bernardo Toninho. 2014. Linear logical relations and observational equivalences for session-based concurrency. *Information and Computation* 239 (2014). https://doi.org/10.1016/j.ic.2014.08.001

Frank Pfenning and Conal Elliott. 1988. Higher-Order Abstract Syntax. In *PLDI*. https://doi.org/10.1145/53990.54010

Benjamin C. Pierce et al. 2020. Programming Language Foundations. https://softwarefoundations.cis.upenn.edu/plf-current/index.html

John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS*. https://doi.org/10.1109/LICS.2002.1029817

Arjen Rouvoet, Casper Bach Poulsen, Robbert Krebbers, and Eelco Visser. 2020. Intrinsically-typed definitional interpreters for linear, session-typed languages. In *CPP*. ACM. https://doi.org/10.1145/3372885.3373818

Ilya Sergey, James R. Wilcox, and Zachary Tatlock. 2018. Programming and proving with distributed protocols. *PACMPL* 2, POPL (2018). https://doi.org/10.1145/3158116

Kasper Svendsen and Lars Birkedal. 2014. Impredicative Concurrent Abstract Predicates. In *ESOP*. https://doi.org/10.1007/978-3-642-54833-8_9

Kasper Svendsen, Lars Birkedal, and Matthew J. Parkinson. 2010. Verifying Generics and Delegates. In *ECOOP*. https://doi.org/10.1007/978-3-642-14107-2_9

Kasper Svendsen, Filip Sieczkowski, and Lars Birkedal. 2016. Transfinite Step-Indexing: Decoupling Concrete and Logical Steps. In *ESOP*. https://doi.org/10.1007/978-3-662-49498-1_28

Nikhil Swamy, Aseem Rastogi, Aymeric Fromherz, Denis Merigoux, Danel Ahman, and Guido Martínez. 2020. SteelCore: an extensible concurrent separation logic for effectful dependently typed programs. *PACMPL* 4, ICFP (2020). https://doi.org/10.1145/3409003

David Swasey, Deepak Garg, and Derek Dreyer. 2017. Robust and compositional verification of object capability patterns. *PACMPL* 1, OOPSLA (2017). https://doi.org/10.1145/3133913

Samira Tasharofi, Peter Dinges, and Ralph E. Johnson. 2013. Why Do Scala Developers Mix the Actor Model with other Concurrency Models?. In *ECOOP*. https://doi.org/10.1007/978-3-642-39038-8_13

Joseph Tassarotti, Ralf Jung, and Robert Harper. 2017. A Higher-Order Logic for Concurrent Termination-Preserving Refinement. In *ESOP (LNCS, Vol. 10201)*. https://doi.org/10.1007/978-3-662-54434-1_34

The CertiK Team. 2021. The CertiK Project. (2021). https://certik.io/

The Coq-std++ Team. 2020. An extended "standard library" for Coq. Available online at https://gitlab.mpi-sws.org/iris/stdpp.

The Erlang Team. 2021. The Erlang Language Project. (2021). https://www.erlang.org/

The Go Team. 2021. The Go Language Project. (2021). https://golang.org/

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

Peter Thiemann and Vasco T. Vasconcelos. 2020. Label-dependent session types. *PACMPL* 4, POPL (2020). https://doi.org/10.1145/3371135

Amin Timany, Léo Stefanesco, Morten Krogh-Jespersen, and Lars Birkedal. 2018. A logical relation for monadic encapsulation of state: Proving contextual equivalences in the presence of runST. *PACMPL* 2, POPL (2018). https://doi.org/10.1145/3158152

Tengfei Tu, Xiaoyu Liu, Linhai Song, and Yiying Zhang. 2019. Understanding Real-World Concurrency Bugs in Go. In *ASPLOS*. https://doi.org/10.1145/3297858.3304069

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

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

Andrew K. Wright. 1995. Simple Imperative Polymorphism. *Lisp and Symbolic Computation* 8, 4 (1995). https://dl.acm.org/doi/10.1007/BF01018828

Andrew K. Wright and Matthias Felleisen. 1994. A Syntactic Approach to Type Soundness. *Information and Computation* 115, 1 (1994). https://doi.org/10.1006/inco.1994.1093