Machine-Checked Semantic Session Typing

Jonas Kastberg Hinrichsen
IT University of Copenhagen, Denmark

Daniël Louwrink
University of Amsterdam, The Netherlands

Robbert Krebbers
Delft University of Technology, The Netherlands

Jesper Bengtson
IT University of Copenhagen, Denmark

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 type safety proof. These extensions cannot be readily combined, and existing type safety proofs are generally not machine-checked, making their correctness less trustworthy. We overcome these shortcomings with a semantic approach to binary asynchronous affine session types using a logical relations model in the Iris framework in Coq. We demonstrate the power of our approach by combining various forms of polymorphism and recursion, asynchronous subtyping, unique and shared references, and locks/mutexes, with little proof effort. As an additional benefit of the semantic approach we demonstrate how to integrate manual verification of racy but safe programs as part of well-typed programs.

1 Introduction

Session types [24] guarantee that message-passing programs comply to 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 they deserve:

1. There exist extensions of session types with *e.g.,* polymorphism [18], asynchronous subtyping [34], and sharing via locks [4]. While type safety has been proven for each extension in isolation, existing proofs cannot be readily composed.
2. Session types use substructural types to enforce a strict discipline of channel ownership. While conventional session-type systems can type-check many useful programs, they inherently exclude some programs that do not obey the ownership discipline, even if they are in fact safe.
3. Only few session-type systems and their safety proofs have been machine-checked by a proof assistant, making their correctness less trustworthy.

To address these challenges, we eschew the traditional *syntactic approach* to type safety (using *progress and preservation*) and embrace the *semantic approach* to type safety [15, 1, 2, 25] (using *logical relations*). In the semantic approach, the typing judgement is an open system (defined in terms of the behaviour of programs), instead of a closed system (defined as an inductive relation with a fixed set of rules). By being an open system, and due to recent developments in program logics [27] and theorem proving [31, 29], we address the above challenges as our system is easily extensible, allows for manual verification of safe programs that do not obey the ownership discipline, and is well-suited for machine-checked proofs.

To concretely see how the semantic approach addresses the first two challenges, let us take a look at the following "racy" program:

$$\lambda\, c.\, (\texttt{recv}\ c \mid\mid \texttt{recv}\ c) \quad : \quad \texttt{chan}\ (?\mathbf{Z}.\, ?\mathbf{Z}.\, \texttt{end}) \multimap (\mathbf{Z} \times \mathbf{Z})$$

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 $\texttt{chan}\ (?\mathbf{Z}.\, ?\mathbf{Z}.\, \texttt{end})$ for a channel that expects to receive two integers). This program cannot be type-checked using conventional session-type systems as channels are substructural/ownership types, and thus cannot be owned by multiple threads at the same time. Nonetheless, 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 to have syntactical typing rules that account for it. Instead, it would be desirable if we could "drop down" to a manual typing proof to show that the program has the ascribed type. Unfortunately, traditional inductively defined typing judgements ($\vdash e : A$) and the progress and preservation proof method cannot account for manual proofs of untyped but safe programs. For the semantic typing judgement ($\vDash e : A$), on the other hand, typing rules (*e.g.,* if $\vDash e_1 : A_1$ and $\vDash e_2 : A_2$, then $\vDash (e_1, e_2) : A_1 \times A_2$) are proven as lemmas instead of being part of the type system's definition. When considering programs whose safety is subtle, one can thus extend the semantic type system by proving additional lemmas like $\vDash \lambda\, c.\, (\texttt{recv}\ c \mid\mid \texttt{recv}\ c) : \texttt{chan}\ (\textbf{?Z}.\, \textbf{?Z}.\, \texttt{end}) \multimap (\mathbf{Z} \times \mathbf{Z})$.

In addition to extending the type system with typing rules, we also want to be able to add type formers (*e.g.,* for polymorphism, recursive types, and locks). We thus follow the *foundational approach* [3, 1] by starting with an untyped language, for which we define types as predicates over values. Type formers are then not part of the definition of the type system either, and can be defined as combinators on these predicates.

Since session types revolve around a discipline of resource ownership, we define semantic types not as set-theoretic predicates, but follow recent work [31, 25, 32, 43, 16] on defining types as predicates in the Iris concurrent separation logic [28, 26, 30, 27]. This way, we can reuse Iris's facilities for dealing with *e.g.,* resource ownership, recursion, polymorphism, and concurrency, as well as its tactical support for proofs in Coq [31, 29].

While Iris provides the means to model standard types, as Iris predicates over values, it does not readily provide the means to model session types. To remedy this shortcoming, we make use of the Actris framework for message-passing in Iris [22]. Actris includes the notion of *dependent separation protocols*, which are similar to session types in structure, and were developed to prove functional correctness of message-passing programs. We show that dependent separation protocols are well-suited for modelling session types semantically, and thus provide a concise connection between session types and separation logic.

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 (§ 2).
- We demonstrate the extensibility of our approach by adding subtyping, copyable types, equi-recursion and polymorphism in term and session types, and mutexes (§ 3).
- We demonstrate the benefit of our type system being open by integrating the manual verification of safe but not type-checkable programs (§ 4).
- We present **Actris 2.0**, which extends Actris with a notion of asynchronous subprotocols. While this notion is designed for functional correctness proofs, it allows us to model the combination of asynchronous subtyping and session type polymorphism (§ 5).
- All of our results are mechanised in Coq (§ 7) and can be found in [23].

[1] For simplicity, we assume that `recv` is atomic, otherwise a lock is needed. Even with a lock, conventional session-type systems cannot handle this program.

Term types:

$$
\begin{aligned}
\mathsf{Type}_{\bigstar} &\triangleq \mathsf{Val} \to \mathsf{iProp}\\
\mathtt{any} &\triangleq \lambda\, w.\, \mathsf{True}\\
\mathbf{1} &\triangleq \lambda\, w.\, w \in \{()\}\\
\mathbf{Z} &\triangleq \lambda\, w.\, w \in \mathbb{Z}\\
\mathbf{B} &\triangleq \lambda\, w.\, w \in \mathbb{B}\\
\mathtt{ref}_{\mathtt{uniq}}\, A &\triangleq \lambda\, w.\, \exists v.\, w \in \mathsf{Loc} * (w \mapsto v) * \triangleright(A\, v)\\
A_1 \times A_2 &\triangleq \lambda\, w.\, \exists w_1, w_2.\, w = (w_1, w_2) *\\
&\qquad\qquad \triangleright(A_1\, w_1) * \triangleright(A_2\, w_2)\\
A_1 + A_2 &\triangleq \lambda\, w.\, \exists v.\, (w = \mathtt{inl}\ v * \triangleright(A_1\, v)) \vee\\
&\qquad\qquad (w = \mathtt{inr}\ v * \triangleright(A_2\, v))\\
A \multimap B &\triangleq \lambda\, w.\, \forall v.\, \triangleright(A\, v) \mathrel{-\!\!*} \mathsf{wp}\ w\ v\ \{B\}\\
\mathtt{chan}\ S &\triangleq \lambda\, w.\, w \rightarrowtail S
\end{aligned}
$$

Judgements:

$$
\begin{aligned}
\Gamma \vDash \sigma &\triangleq \mathop{\Large\ast}_{(x,A)\in\Gamma}.\ \exists v.\, (x, v) \in \sigma * A\, v\\
\Gamma \vDash e : A \dashv \Gamma' &\triangleq \forall \sigma.\, (\Gamma \vDash \sigma) \mathrel{-\!\!*}\\
&\qquad \mathsf{wp}\ e[\sigma]\ \{v. A\, v * (\Gamma' \vDash \sigma)\}\\
\Gamma \bumpeq \Gamma_1 \circ \Gamma_2 &\triangleq \forall \sigma.\, (\Gamma \vDash \sigma) \mathrel{*\!\!-\!\!*} (\Gamma_1 \vDash \sigma * \Gamma_2 \vDash \sigma)
\end{aligned}
$$

Session types:

$$
\begin{aligned}
\mathsf{Type}_{\blacklozenge} &\triangleq \mathsf{iProto}\\
\mathsf{end} &\triangleq \mathbf{end}\\
\mathbf{!}A.\, S &\triangleq \mathbf{!}\,(v : \mathsf{Val})\, \langle v \rangle \{\triangleright(A\, v)\}.\, S\\
\mathbf{?}A.\, S &\triangleq \mathbf{?}\,(v : \mathsf{Val})\, \langle v \rangle \{\triangleright(A\, v)\}.\, S\\
\oplus\{\vec{S}\} &\triangleq \mathbf{!}\,(l : \mathbb{Z})\, \langle l \rangle \big\{\triangleright(l \in \mathrm{dom}(\vec{S}))\big\}.\, \vec{S}(l)\\
\&\{\vec{S}\} &\triangleq \mathbf{?}\,(l : \mathbb{Z})\, \langle l \rangle \big\{\triangleright(l \in \mathrm{dom}(\vec{S}))\big\}.\, \vec{S}(l)
\end{aligned}
$$

Figure 1 Semantic typing for simply-typed session types.

2 Tour of the Semantic Session Typing

We give a tour of the semantic approach by developing a semantic type system using logical relations for the simply-typed fragment of a session-type system. Our development follows the standard approach to logical relations in Iris [31, 25, 32, 43, 16], but we take the *foundational approach* to semantic type safety [3, 1] to the full extent. In particular, we do not define a syntactic type system and give an interpretation of that, but define type formers simply as combinators on semantic types (§ 2.1) and session types (§ 2.2). This approach gives rise to an open type system that can easily be extended with new type formers (§ 3), and it eases mechanisation as we can use Coq's meta-level binding for binding in types (§ 7).

We use an untyped higher-order functional programming language with concurrency, unique references, and binary asynchronous message passing:

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

We omit the standard operations on pairs, sums, *etc.* We write $\lambda\, x.\, e$ for $\mathtt{rec}\, \_(x) = e$, and $\mathtt{let}\ x = e_1\ \mathtt{in}\, e_2$ for $(\lambda\, x.\, e_2)\ e_1$, and $e_1; e_2$ for $\mathtt{let}\ \_ = e_1\ \mathtt{in}\, e_2$. Message passing is given an asynchronous semantics: $\mathtt{new\_chan}\ ()$ returns a pair (c_1, c_2) of channel endpoints that operate on buffers $(\vec{v}_1, \vec{v}_2)$ that are initially empty, $\mathtt{send}\ c_i\ w$ enqueues message w in $\vec{v}_i$, while $\mathtt{recv}\ c_i$ 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. Parallel composition $e_1 \mid\mid e_2$ executes e_1 and e_2 in parallel and returns the results as a tuple. The language also supports fork and compare-and-swap.

2.1 Term Types

The definitions are shown in Figure 1. Types Type_k are indexed by kinds; $\bigstar$ for *term types* and $\blacklozenge$ 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 $\mathsf{Type}_{\bigstar} \triangleq \mathsf{Val} \to \mathsf{iProp}$ are defined as Iris predicates over values, where iProp is the type of Iris propositions. Session types $\mathsf{Type}_{\blacklozenge} \triangleq \mathsf{iProto}$ are defined as dependent separation protocols of Actris (§ 2.2).

Type Formers The ground types (the unit type $\mathbf{1}$, Boolean type $\mathbf{B}$, and integer type $\mathbf{Z}$) are defined through membership of the corresponding set ($\{()\}$, $\mathbb{B}$, and $\mathbb{Z}$, respectively). The type former $\texttt{ref}_{\texttt{uniq}}\, A$ for uniquely owned references, and the type former $A_1 \times A_2$ for products nicely demonstrate the advantage of separation logic—since types ($\mathsf{Type}_\bigstar \triangleq \mathsf{Val} \to \mathsf{iProp}$) are not set-theoretic predicates, but Iris predicates, they do not just describe knowledge, but also cover the substructural aspects of session types by describing which resources are owned. We make use of the *points-to connective* $w \mapsto v$ of separation logic to describe that $\texttt{ref}_{\texttt{uniq}}\, A$ consists of the locations $w \in \mathsf{Loc}$ that hold a value v for which the resources $A\, v$ are owned. We make use of the *separating conjunction* ($*$) 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*. In addition to the aforementioned standard connectives of separation logic, we use Iris's *later modality* ($\triangleright$) to ensure that type formers are *contractive*, which is needed to model equi-recursive types as we further detail in § 3.

Another type former that shows the benefits of separation logic is the function type $A \multimap B$. It describes the values w that when applied to an argument consume the resources A, and in return, produce the resources B. The definition $\triangleright(A\, v) \mathrel{-\!\!*} \mathsf{wp}\ w\ v\ \{B\}$ makes use of the *magic wand* ($\mathrel{-\!\!*}$) of separation logic and the *weakest precondition* connective ($\mathsf{wp}\ e\ \{\Phi\}$) of Iris.[2] The weakest precondition $\mathsf{wp}\ e\ \{\Phi\}$ dictates that given a postcondition $\Phi : \mathsf{Val} \to \mathsf{iProp}$, the expression e is safe, and if it reduces to a value v, then $\Phi\, v$ holds.

It is important to stress that none of the above definitions involve explicit accounting of resources—all of that is handled implicitly by working in separation logic.

Typing Judgement Since the types of references and channels can change via strong updates, we use a typing judgement $\Gamma \vDash e : A \dashv \Gamma'$ (defined in Figure 1) with a *pre-* and *post-context* $\Gamma, \Gamma' \in \mathsf{String} \rightharpoonup \mathsf{Type}_\bigstar$, which describe the types of variables before and after execution of e. As is standard in logical relations, we use *closing substitutions* $\sigma \in \mathsf{String} \rightharpoonup \mathsf{Val}$ and an auxiliary judgement $\Gamma \vDash \sigma$. The definition of this judgement employs the iterated separation conjunction $\mathop{\Huge *}_{(x,A)\in\Gamma}$ to ensure that for each variable binding (x, A) in Γ there is a binding (x, v) in σ for which the resources $A\, v$ are owned separately. The typing judgement is then 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'.

The *splitting relation* $\Gamma \bumpeq \Gamma_1 \circ \Gamma_2$ states that the resources governed by the context Γ can be split into contexts Γ_1 and Γ_2. It is defined semantically (using the *bi-wand* $\mathrel{*\!\!-\!\!*}$ and $\Gamma \vDash \sigma$ judgement), and it enjoys the usual splitting rules as lemmas. We often inline $\Gamma_1 \circ \Gamma_2$ to denote the existence of a context Γ for which $\Gamma \bumpeq \Gamma_1 \circ \Gamma_2$ holds.

Typing Rules Now that the type formers and the typing judgement are in place, we can prove each typing rule by unfolding the definition of the semantic typing judgement $\Gamma \vDash e : A \dashv \Gamma'$ and proving the corresponding proposition in Iris. Examples of typing rules are:

$$\Gamma, (x : A) \vDash x : A \dashv \Gamma, (x : \texttt{any}) \qquad \Gamma, (x : \texttt{ref}_{\texttt{uniq}}\, A) \vDash\, !\, x : A \dashv \Gamma, (x : \texttt{ref}_{\texttt{uniq}}\, \texttt{any})$$

$$\frac{\Gamma_1 \vDash e_1 : A \dashv \Gamma_2 \qquad \Gamma_2, (x : A) \vDash e_2 : B \dashv \Gamma_3}{\Gamma_1 \vDash (\texttt{let}\, x = e_1\, \texttt{in}\, e_2) : B \dashv \Gamma_3 \setminus x} \qquad \frac{\Gamma_1 \vDash e_1 : A_1 \dashv \Gamma_1' \qquad \Gamma_2 \vDash e_2 : A_2 \dashv \Gamma_2'}{\Gamma_1 \circ \Gamma_2 \vDash e_1 \mid\mid e_2 : A_1 \times A_2 \dashv \Gamma_1' \circ \Gamma_2'}$$

[2] Weakest preconditions $\mathsf{wp}\ e\ \{\Phi\}$ are contractive in Φ if e is not a value. As such, no later modality is needed around B in the definition of $A \multimap B$.

Actris dependent separation protocol rules for channels:

$$\begin{aligned}
&\textsf{wp}\ \texttt{new\_chan}\ ()\ \{(c,c').\ c \rightarrowtail \mathit{prot} * c' \rightarrowtail \overline{\mathit{prot}}\}\\
c \rightarrowtail \mathbf{!}\,\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot} * \triangleright P[\vec{t}/\vec{x}] \mathrel{-\!\!*}\ &\textsf{wp}\ \texttt{send}\ c\ (v[\vec{t}/\vec{x}])\ \{c \rightarrowtail \mathit{prot}[\vec{t}/\vec{x}]\}\\
c \rightarrowtail \mathbf{?}\,\vec{x}:\vec{\tau}\,\langle v\rangle\{\triangleright P\}.\,\mathit{prot} \mathrel{-\!\!*}\ &\textsf{wp}\ \texttt{recv}\ c\,\{w.\,\exists\vec{y}.\,(w = v[\vec{y}/\vec{x}]) * c \rightarrowtail \mathit{prot}[\vec{y}/\vec{x}] * P[\vec{y}/\vec{x}]\}
\end{aligned}$$

Semantic typing rules for channels:

$$\Gamma \vDash \texttt{new\_chan}\ () : \texttt{chan}\ S \times \texttt{chan}\ \overline{S} \dashv \Gamma$$

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

$$\frac{1 \le i \le n}{\Gamma, (x : \texttt{chan}\ (\oplus\{l_1 : S_1, \ldots, l_n : S_n\})) \vDash \texttt{select}\ x\ l_i : \mathbf{1} \dashv \Gamma, (x : \texttt{chan}\ S_i)}$$

$$\frac{\Gamma, (x : \texttt{chan}\ S_1) \vDash e_1 : A \dashv \Gamma' \quad \cdots \quad \Gamma, (x : \texttt{chan}\ S_n) \vDash e_n : A \dashv \Gamma'}{\Gamma, (x : \texttt{chan}\ (\&\{l_1 : S_1, \ldots, l_n : S_n\})) \vDash \texttt{branch}\ x\ \texttt{with}\ l_1 \Rightarrow e_1 \mid \ldots \mid l_n \Rightarrow e_n : A \dashv \Gamma'}$$

Figure 2 The Actris and semantic typing rules for channels.

Since we are working in a substructural type system, reading from a variable or loading from a location causes *ownership to be moved out*, *i.e.,* the type becomes $\texttt{any}$, which asserts no ownership. The above rules also move out ground types ($\mathbf{1}$, $\mathbf{B}$, and $\mathbf{Z}$), we address how these can be retained in § 3. To account for strong-updates, the rule for let-expressions threads contexts through the constituents. The variable x is added to the pre-context Γ_2 of e_2, and removed from Γ_3 to ensure that it does not escape the scope of e_2. The parallel composition rule states that the pre-context needs to be split into disjoint parts Γ_1 and Γ_2, which are moved to their respective threads. The post-contexts Γ'_1 and Γ'_2 are joined in the end.

Since the proofs of typing these rules are carried out in Iris, which implicitly takes care of resource accounting by being a separation logic, they typically involve just a couple of lines of Coq code in the mechanisation thanks to Iris's tactics for separation logic [31, 29].

Type Safety Type safety means: if $\varnothing \vDash e : A \dashv \Gamma$, then e is safe, *i.e.,* e will not get stuck w.r.t. the operational semantics. For syntactic type systems, type safety is usually proven via 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 [30, 27]. Note that our type system is affine (resources are not explicitly deallocated), and thus the post-context Γ need not be empty. We use an affine type system because that allows more practical programs to be typeable, while giving the desired safety result.

2.2 Session Types

Now that the core semantic type system is in place, we extend it with session types (*i.e.,* protocols). In this section we consider the basic type formers for sending a message $\mathbf{!}A.\,S$, receiving a message $\mathbf{?}A.\,S$, the choice primitives for selection $\oplus\{\vec{S}\}$ and branching $\&\{\vec{S}\}$, and the terminator $\texttt{end}$. We let $\vec{S} : \mathbb{Z} \rightharpoonup \textsf{Type}_{\blacklozenge}$, and often write $\vec{S} = l_1 : S_1, \ldots l_n : S_n$. The term type $\texttt{chan}\ S$ dictates that a term is a channel that follows the session type S.

Dependent Separation Protocols Session types are defined in terms of Actris's *dependent separation protocols* [22], which are similar to session types in structure, but can express functional properties of the transferred data. Dependent separation protocols $\mathit{prot} \in \mathsf{iProto}$ 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. 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. Finite protocols are ultimately terminated by an **end** constructor. As an example, the dependent separation protocol $\mathbf{!}\,(\ell : \mathsf{Loc})\,(i : \mathbb{Z})\,\langle \ell\rangle\{\ell \mapsto i * 10 \leq i\}.\,\mathbf{?}\,\langle ()\rangle\{\ell \mapsto (i+1)\}.\,\mathbf{end}$ 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.

The construct $c \rightarrowtail \mathit{prot}$ denotes ownership of a channel c with a dependent separation protocol prot. The Actris proof rules are shown in Figure 2. The rule for `new_chan` $()$ allows ascribing any protocol to a newly created channel, obtaining ownership of $c \rightarrowtail \mathit{prot}$ and $c' \rightarrowtail \overline{\mathit{prot}}$ for the respective endpoints. Here, $\overline{\mathit{prot}}$ is the *dual* of prot in which any receive ($\mathbf{?}$) is turned into a send ($\mathbf{!}$), and *vice versa*. The rule for `send` $c\ w$ requires the head of the protocol to be a send ($\mathbf{!}$), 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 $c \rightarrowtail \mathbf{!}\,\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot}$, pick an appropriate instantiation $\vec{t}$ for the variables $\vec{x}{:}\vec{\tau}$ so that $w = v[\vec{t}/\vec{x}]$, and give up ownership of the associated resources $P[\vec{t}/\vec{x}]$. Subsequently, one gets back ownership of the protocol tail $\mathit{prot}[\vec{t}/\vec{x}]$. The rule for `recv` c is essentially dual to the rule for `send` $c\ w$. One needs to give up ownership of $c \rightarrowtail \mathbf{?}\,\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot}$, 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 $\mathit{prot}[\vec{y}/\vec{x}]$, where $\vec{y}$ are instances of the variables of the protocol.

Semantics of Session Types The definitions of session types are shown in Figure 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 $w \rightarrowtail S$, and duality $\overline{S}$ of session types S is inherited from Actris. The definition of the terminator (`end`), send ($\mathbf{!}$), and receive ($\mathbf{?}$) follow from their dependent separation protocol counterparts. For example $\mathbf{!}A.\,S$ is defined as $\mathbf{!}\,(v : \mathsf{Val})\,\langle v\rangle\{\triangleright(A\ v)\}.\,S$. It says that a value v is sent along with ownership of $A\ v$. Similar to term type formers, the later modality ($\triangleright$) is put in place to achieve contractiveness, needed for equi-recursive types (§ 3).

While the choice types $\oplus\{\vec{S}\}$ and $\&\{\vec{S}\}$ do not have a direct dependent separation protocol counterpart, they can be defined using send and receive. For example, $\oplus\{\vec{S}\}$ is defined as $\mathbf{!}\,(l : \mathbb{Z})\,\langle l\rangle\{\triangleright(l \in \mathrm{dom}(\vec{S}))\}.\,\vec{S}(l)$. It expresses that a valid label $l \in \mathrm{dom}(\vec{S})$ (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.

Session Typing Rules The session typing rules are shown in Figure 2. 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:

$$\texttt{select}\ x\ l \triangleq \texttt{send}\ x\ l$$

$$\begin{aligned}\texttt{branch}\ x\ \texttt{with}\ l_1 \Rightarrow e_1 \mid \ldots \mid l_n \Rightarrow e_n \triangleq{}& \texttt{let}\ y = \texttt{recv}\ x\ \texttt{in}\ \texttt{if}\ y = l_1\ \texttt{then}\ e_1\ \texttt{else}\ \cdots\\ & \texttt{if}\ y = l_n\ \texttt{then}\ e_n\ \texttt{else}\ ()\,()\end{aligned}$$

Safety of the branch operation guarantees that the stuck expression $()\,()$ is never executed, and hence, one of the branches is always found.

3 Extending the Type System

We demonstrate the extensibility of our approach to session types by adding subtyping, copyable types, equi-recursion, polymorphism, and locks/mutexes.

Term-Level Subtyping Subtyping $A <: B$ indicates that any member of type A is also a member of type B. As stronger type predicates permit fewer members than weaker ones, we define the semantic subtyping relation as the point-wise lifting of the magic wand:

$$A <: B \triangleq \forall v.\, A\, v \mathrel{-\!\!*} B\, v$$

Since Iris weakest preconditions are closed under magic wands, *i.e.,* they enjoy the proof rule $(\forall v.\, \Phi_1\, v \mathrel{-\!\!*} \Phi_2\, v) \mathrel{-\!\!*} \mathsf{wp}\; e\, \{\Phi_1\} \mathrel{-\!\!*} \mathsf{wp}\; e\, \{\Phi_2\}$, no change to the definition of the semantic typing judgement $\Gamma \vDash e : B \dashv \Gamma'$ (defined in Figure 1) is needed. We prove the subsumption rule together with the conventional subtyping rules as mere lemmas. For example:

$$\frac{A <: B \qquad \Gamma \vDash e : A \dashv \Gamma'}{\Gamma \vDash e : B \dashv \Gamma'} \qquad A <: A \qquad \frac{A <: B \qquad B <: C}{A <: C} \qquad \frac{A <: B}{\texttt{ref}_{\texttt{uniq}}\, A <: \texttt{ref}_{\texttt{uniq}}\, B}$$

Although the type formers we defined so far do not enjoy any non-structural subtyping rules apart from $A <: \texttt{any}$, we prove more rules throughout the paper while extending the type system. In § 5 we add subtyping for session types as well.

Copyable Types and Shared References As session-type systems are substructural, types denote ownership and can be used only once. This becomes evident by the variable and load rules from § 2.1, which *move out ownership* by turning the element type into $\texttt{any}$. While moving out ownership is necessary for soundness in general, this is too restrictive for types such as $\mathbf{B}$, $\mathbf{Z}$, $\mathbf{Z} * \mathbf{B}$ that do not assert ownership of any resources, and need not be moved out. We therefore extend the type system with a notion of *copyable types*. Concretely, we define type formers $\texttt{copy}$ and $\overline{\texttt{copy}}$, and a property $\texttt{copyable}$:

$$\texttt{copy}\, A \triangleq \lambda\, w.\, \Box(A\, w) \qquad \texttt{copyable}\, A \triangleq A <: \texttt{copy}\, A \qquad \overline{\texttt{copy}}\, A \triangleq \lambda\, w.\, \texttt{coreP}\; (A\, w)$$

The type former $\texttt{copy}$ uses the $\Box$ modality of Iris, which makes a proposition *persistent*. Persistent propositions can be duplicated freely by the proof rule $\Box\, P \mathrel{-\!\!*} (\Box\, P * P)$. The property $\texttt{copyable}\, A$ simply states that A can be subtyped into $\texttt{copy}\, A$, and thus copyable types can be used without being moved. Ground types ($\mathbf{1}$, $\mathbf{B}$, $\mathbf{Z}$) and $\texttt{copy}$ are copyable, and copyable is closed under products and sums. Note that copyable is not closed under functions $A \multimap B$, as they can capture resources from the context. We therefore define $A \to B \triangleq \texttt{copy}\, (A \multimap B)$ for the copyable (*i.e.,* unrestricted) function type.

The type former $\overline{\texttt{copy}}$ is the inverse of $\texttt{copy}$. It is defined using Iris's $\texttt{coreP}$ modality, which is persistent, and enjoys the rule $\texttt{coreP}\; (\Box\, P) \mathrel{-\!\!*} P$. This gives that $\overline{\texttt{copy}}$ is copyable, as well as the subtyping $\overline{\texttt{copy}}\, (\texttt{copy}\, A) <: A$. We use $\overline{\texttt{copy}}$ to formulate more flexible rules for variables, load, and other operators that move out ownership:

$$\Gamma, (x : A) \vDash x : A \dashv \Gamma, (x : \overline{\texttt{copy}}\, A) \qquad \Gamma, (x : \texttt{ref}_{\texttt{uniq}}\, A) \vDash\, !\, x : A \dashv \Gamma, (x : \texttt{ref}_{\texttt{uniq}}\, (\overline{\texttt{copy}}\, A))$$

These rules are strictly more general than the rules in § 2.1, as we can always subtype to $\texttt{any}$. However, if A is copyable, then $\overline{\texttt{copy}}\, A <: A$, meaning we can avoid moving out ownership. Making use of the fact that the splitting judgement $(\Gamma \bumpeq \Gamma_1 \circ \Gamma_2)$ is defined semantically, we extend it by proving a lemma that allows variables of copyable type to be used in Γ_1 and Γ_2.

Another example of a copyable type that can be added to our type system is that of shared references $\texttt{ref}_{\texttt{shr}}\, A$, whose definition is $\texttt{ref}_{\texttt{shr}}\, A \triangleq \lambda\, w.\, (w \in \mathsf{Loc}) * \boxed{\exists v.\, (w \mapsto v) * \Box(A\, v)}$. This definition makes use of Iris invariants $\boxed{P}$, which state that P holds at all times. As the definition is standard for Iris-style logical relations it is out of scope for this paper.

Equi-Recursive Term and Session Types We extend the type system with equi-recursive types using Iris's operator $\mu\,(x : \tau).\, t$ for *guarded recursion* [35]. The operator is *guarded* as it requires the variable x to appear in a *contractive* or *guarded* position, *e.g.,* in the argument P of the later modality $\triangleright P$, the postcondition Φ of a weakest precondition $\mathsf{wp}\ e\ \{\Phi\}$, or the tail *prot* of a protocol $\mathbf{!}\,\vec{x}\!:\!\vec{\tau}\,\langle v\rangle\{P\}.\, \mathit{prot}$ or $\mathbf{?}\,\vec{x}\!:\!\vec{\tau}\,\langle v\rangle\{P\}.\, \mathit{prot}$. We lift the guarded recursion operator of Iris into a *kinded* operator for equi-recursion in the type system:

$$\mu\,(X : k).\, K \triangleq \mu\,(X : \mathsf{Type}_k).\, K \qquad (K \text{ must be contractive in } X)$$

To utilise equi-recursion, we carefully put later modalities in the definitions of the type formers to ensure they are contractive in all arguments. As such, we can construct arbitrary recursive term and session types, including examples from the session type literature [20], such as $\mu\,(X : \blacklozenge).\, \mathbf{!}(\texttt{chan}\ X).\, X$, 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. Instead of putting later modalities in the definitions of type formers, they put a later modality in the definition of the recursion operator. This avoids the contractive side-condition, but requires "fold" and "unfold" operators in the language.

Polymorphism in Term and Session Types Following other logical relation developments in Iris, we make use of Iris's logical quantifiers to extend our type system with kinded polymorphism at the term level:

$$\begin{aligned}
&\forall (X : k).\, A \triangleq \lambda\, w.\, \forall (X : \mathsf{Type}_k).\, \mathsf{wp}\ w\ ()\ \{A\}\\
&\exists (X : k).\, A \triangleq \lambda\, w.\, \exists (X : \mathsf{Type}_k).\ \triangleright(A\, w)
\end{aligned}$$

These types are defined in the style of parametricity—they quantify over semantic types $X : \mathsf{Type}_k$. This is possible because Iris supports higher-order impredicative quantification (*i.e.,* quantification over Iris predicates). Values of the universal types are thunks (if $\vDash e : A$, then $\vDash \lambda\_.\, e : \forall X.\, A$) as shown by the weakest precondition in the definition. The typing rules for term-level polymorphism are standard and thus elided.

A more interesting extension is polymorphism in session types [18]. A simple example is an interaction with a polymorphic "mapper" service $\mu\,(\mathit{rec} : \blacklozenge).\, \mathbf{!}_{(X,Y:\bigstar)}\,(X \multimap Y).\, \mathbf{!}X.\, \mathbf{?}Y.\, \mathit{rec}$ that allows sending a function $X \multimap Y$ and an input X, and in return receive the output Y. Different types can be picked for the type variables X and Y 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:

$$\begin{aligned}
&\mathbf{!}_{\vec{X}:\vec{k}}\, A.\, S \triangleq \mathbf{!}\,(\vec{X} : \vec{\mathsf{Type}_k})(v : \mathsf{Val})\,\langle v\rangle\{\triangleright(A\, v)\}.\, S\\
&\mathbf{?}_{\vec{X}:\vec{k}}\, A.\, S \triangleq \mathbf{?}\,(\vec{X} : \vec{\mathsf{Type}_k})(v : \mathsf{Val})\,\langle v\rangle\{\triangleright(A\, v)\}.\, S
\end{aligned}$$

This definition relies on the fact that binders $\vec{x}\!:\!\vec{\tau}$ in Actris's dependent separation protocols $\mathbf{!}\,\vec{x}\!:\!\vec{\tau}\,\langle v\rangle\{P\}.\, \mathit{prot}$ and $\mathbf{?}\,\vec{x}\!:\!\vec{\tau}\,\langle v\rangle\{P\}.\, \mathit{prot}$ are higher-order and impredicative.

The typing rules are extended to allow instantiation of binders when sending a message and elimination of type variables when receiving a message. The rule for receive becomes:

$$\frac{\Gamma, (x : \texttt{chan}\ S), (y : A) \vDash e : B \dashv \Gamma' \qquad \vec{X} \text{ free in } \Gamma,\, \Gamma', \text{ and } B}{\Gamma, (x : \texttt{chan}\ (\mathbf{?}_{\vec{X}:\vec{k}}\, A.\, S)) \vDash \texttt{let}\ y = \texttt{recv}\ x\ \texttt{in}\ e : B \dashv \Gamma' \setminus \{x\}}$$

This rule requires the result y of `recv` to be let-bound to ensure that the type variables $\vec{X}$ cannot escape into Γ' or B.

$$R \mathrel{-\!\!*} \mathsf{wp}\ \texttt{newlock}\ ()\ \{lk.\ \texttt{isLock}\ lk\ R\} \qquad \Gamma, (x : A) \vDash \mathit{newmutex}\ x : \texttt{mutex}\ A \dashv \Gamma$$

$$\texttt{isLock}\ lk\ R \mathrel{-\!\!*} \mathsf{wp}\ \texttt{acquire}\ lk\ \{R\} \qquad \Gamma, (x : \texttt{mutex}\ A) \vDash \mathit{acquire}\ x : A \dashv \Gamma, (x : \overline{\texttt{mutex}}\ A)$$

$$\texttt{isLock}\ lk\ R * R \mathrel{-\!\!*} \mathsf{wp}\ \texttt{release}\ lk\ \{\mathsf{True}\} \qquad \frac{\Gamma \vDash e : A \dashv \Gamma', (x : \overline{\texttt{mutex}}\ A)}{\Gamma \vDash \mathit{release}\ x\ e : \mathbf{1} \dashv \Gamma', (x : \texttt{mutex}\ A)}$$

Figure 3 The Iris proof rules and semantic typing rules for locks/mutexes.

Locks and Mutexes Since channels (of type $\texttt{chan}\ S$) are substructural, they can be used by at most one thread at the same time. Balzer *et al.* [4] 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 $\texttt{mutex}\ A$ 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 "mapper" service from the previous section between multiple clients—they can acquire the mutex, request any number of mapping actions, retrieve the corresponding results, and then release the mutex:

$$\texttt{mutex}\ \big(\mu\,(rec : \blacklozenge).\ \mathbf{!}_{(X, Y : \bigstar)}\ (X \multimap Y).\ \mathbf{!}X.\ \mathbf{?}Y.\ rec\big)$$

The $\texttt{mutex}$ type former is copyable, and comes with operations *newmutex* to allocate a mutex, *acquire* to acquire a mutex by blocking until no other thread holds it, and *release* to release the mutex. The typing rules are shown in Figure 3 and include the type former $\overline{\texttt{mutex}}$, which signifies that the mutex is acquired.

To extend our type system with mutexes we make use of the locks library that is available in Iris. This library consists of operations $\texttt{newlock}$, $\texttt{acquire}$, and $\texttt{release}$, which are similar to the mutex operations, but do not protect a value. The mutex operations are defined in terms of locks as follows:

$$\begin{aligned} \mathit{newmutex} &\triangleq \lambda\, y.\ (\texttt{newlock}\ (), \texttt{ref}\ y) \\ \mathit{acquire} &\triangleq \lambda\, x.\ \texttt{acquire}\ (\texttt{fst}\ x);\ !\,(\texttt{snd}\ x) \\ \mathit{release} &\triangleq \lambda\, x\ y.\ (\texttt{snd}\ x) \leftarrow y;\ \texttt{release}\ (\texttt{fst}\ x) \end{aligned}$$

That is, *newmutex* creates a lock alongside a boxed value. The value can then be acquired with *acquire*, which first acquires the lock. Finally, *release* moves the value back into the box, and releases the lock.

The Iris rules for locks are shown in Figure 3 and make use of the representation predicate $\texttt{isLock}\ lk\ R$, 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 $\texttt{isLock}\ lk\ R$. 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 the following type formers for mutexes:

$$\begin{aligned} \texttt{mutex}\ A &\triangleq \lambda\, w.\ \exists lk, \ell.\ (w = (lk, \ell)) * \texttt{isLock}\ lk\ (\exists v.\ (\ell \mapsto u) * \triangleright(A\ v)) \\ \overline{\texttt{mutex}}\ A &\triangleq \lambda\, w.\ \exists lk, \ell.\ (w = (lk, \ell)) * \texttt{isLock}\ lk\ (\exists v.\ (\ell \mapsto u) * \triangleright(A\ v)) * (\ell \mapsto -) \end{aligned}$$

The $\texttt{mutex}$ 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 3 are proven as lemmas.

4 Manual Typing Proofs

We demonstrate how programs that are not typeable using the typing rules can be typed via a manual proof in Iris/Actris. We call such proofs *manual typing proofs*. Consider the example from the introduction (where the locks have been made explicit):

$$\begin{aligned} \mathit{threadprog} &\triangleq \lambda\, c\ \mathit{lk}.\, \texttt{acquire}\ \mathit{lk};\ \texttt{let}\ x = \texttt{recv}\ c\ \texttt{in}\ \texttt{release}\ \mathit{lk};\ x \\ \mathit{lockprog} &\triangleq \lambda\, c.\, \texttt{let}\ \mathit{lk} = \texttt{newlock}\ ()\ \texttt{in}\ (\mathit{threadprog}\ c\ \mathit{lk} \mid\mid \mathit{threadprog}\ c\ \mathit{lk}) \end{aligned}$$

We would like to prove $\vDash \mathit{lockprog} : \texttt{chan}\ (\mathbf{?Z}.\, \mathbf{?Z}.\, \texttt{end}) \multimap (\mathbf{Z} \times \mathbf{Z})$. This typing judgement is not derivable from the typing rules we presented so far, even if we were to use mutexes instead of plain locks. After all, the channel type changes each time the lock/mutex is required and released. Fortunately, we can unfold the definition of the semantic typing judgement, which gives us the following proof obligation in Iris/Actris:

$$\begin{aligned} &(c \rightarrowtail \mathbf{?}\,(v_1 : \mathsf{Val})\ \langle v_1 \rangle \{\triangleright(v_1 \in \mathbb{Z})\}.\, \mathbf{?}\,(v_2 : \mathsf{Val})\ \langle v_2 \rangle \{\triangleright(v_2 \in \mathbb{Z})\}.\, \mathbf{end}) \mathrel{-\!\!*} \\ &\mathsf{wp}\ \mathit{lockprog}\ c\ \{v.\, \exists v_1, v_2.\, (v = (v_1, v_2)) * \triangleright(v_1 \in \mathbb{Z}) * \triangleright(v_2 \in \mathbb{Z})\} \end{aligned}$$

The proof is carried out using Iris's *ghost state* machinery. We use a *fractional* ghost resource $\boxed{q}^{\gamma}$ with $q \in (0,1]_{\mathbb{Q}}$, which reflects how much of the channel protocol its owner is allowed to resolve, as enforced by the following lock invariant:

$$\begin{aligned} \texttt{chaninv} \triangleq\ &(c \rightarrowtail \mathbf{?}\,(v_1 : \mathsf{Val})\ \langle v_1 \rangle \{\triangleright(v_1 \in \mathbb{Z})\}.\, \mathbf{?}\,(v_2 : \mathsf{Val})\ \langle v_2 \rangle \{\triangleright(v_2 \in \mathbb{Z})\}.\, \mathbf{end}) && \vee \quad (1) \\ &(c \rightarrowtail \mathbf{?}\,(v_2 : \mathsf{Val})\ \langle v_2 \rangle \{\triangleright(v_2 \in \mathbb{Z})\}.\, \mathbf{end}) * \boxed{1/2}^{\gamma} && \vee \quad (2) \\ &(c \rightarrowtail \mathbf{end}) * \boxed{1}^{\gamma} && \quad\ \ (3) \end{aligned}$$

This invariant describes that the channel is in one of three states: (1) no values have been received yet, (2) one value has been received, or (3) all values have been received. State (2) and (3) assert half and full ownership of the ghost resource, respectively to adequately require giving up ones capability of taking a step when performing a receive.

The proof is carried out by allocating full ownership of a ghost resource $\boxed{1}^{\gamma}$ (with a fresh ghost variable name γ), after which the lock predicate $\texttt{isLock}\ \mathit{lk}\ \texttt{chaninv}$ is allocated by giving up full ownership of the channel c. The ghost resource is split into two halves $\boxed{1/2}^{\gamma}$, which are each delegated to a thread, along with the persistent lock predicate $\texttt{isLock}\ \mathit{lk}\ \texttt{chaninv}$. Both threads have the same proof obligation, namely showing that they return an integer:

$$\left(\texttt{isLock}\ \mathit{lk}\ \texttt{chaninv} * \boxed{1/2}^{\gamma}\right) \mathrel{-\!\!*} \mathsf{wp}\ \mathit{threadprog}\ c\ \mathit{lk}\ \{v.\, v \in \mathbb{Z}\}$$

The verification is carried out as follows. First, the lock invariant is obtaining by acquiring the lock. The channel can either be in state (1) or (2), as having half of the ownership excludes the possibility of the full ownership being in the lock (and thereby state (3)). Regardless of being in state (1) or (2), the thread takes a step of the protocol by receiving the value, and then releases the lock by giving up ownership of the updated channel along with their ownership fragment, giving up their capability of taking another step.

5 Actris 2.0: Subprotocols

As demonstrated in § 3, a benefit of the semantic approach is that existing features of Iris and Actris can be used to extend the type system. We now extend Actris with a new feature of *subprotocols* that lets us extend the type system with a rich notion of protocol subtyping. We call the extension of Actris with subprotocols **Actris 2.0**.

Actris 2.0 subprotocol rules:

$$
\begin{aligned}
(\mathit{prot}_1 \sqsubseteq \mathit{prot}_2) &\mathrel{-\!\!*} (c \rightarrowtail \mathit{prot}_1) \mathrel{-\!\!*} (c \rightarrowtail \mathit{prot}_2) && (1)\\
\triangleright(\mathit{prot}_1 \sqsubseteq \mathit{prot}_2) &\mathrel{-\!\!*} (\mathbf{!}\,\langle v\rangle\{P\}.\,\mathit{prot}_1 \sqsubseteq \mathbf{!}\,\langle v\rangle\{P\}.\,\mathit{prot}_2) && (2)\\
\triangleright(\mathit{prot}_1 \sqsubseteq \mathit{prot}_2) &\mathrel{-\!\!*} (\mathbf{?}\,\langle v\rangle\{P\}.\,\mathit{prot}_1 \sqsubseteq \mathbf{?}\,\langle v\rangle\{P\}.\,\mathit{prot}_2) && (3)\\
\mathbf{?}\,\langle v_1\rangle\{P_1\}.\,\mathbf{!}\,\langle v_2\rangle\{P_2\}.\,\mathit{prot} &\sqsubseteq \mathbf{!}\,\langle v_2\rangle\{P_2\}.\,\mathbf{?}\,\langle v_1\rangle\{P_1\}.\,\mathit{prot} && (4)\\
P[\vec{t}/\vec{x}] \mathrel{-\!\!*} \big(\mathbf{!}\,\vec{x}\!:\!\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot} &\sqsubseteq \mathbf{!}\,\langle v[\vec{t}/\vec{x}]\rangle\{\mathsf{True}\}.\,\mathit{prot}[\vec{t}/\vec{x}]\big) && (5)\\
P[\vec{t}/\vec{x}] \mathrel{-\!\!*} \big(\mathbf{?}\,\langle v[\vec{t}/\vec{x}]\rangle\{\mathsf{True}\}.\,\mathit{prot}[\vec{t}/\vec{x}] &\sqsubseteq \mathbf{?}\,\vec{x}\!:\!\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot}\big) && (6)\\
\big(\forall \vec{x}\!:\!\vec{\tau}.\,P \mathrel{-\!\!*} \big(\widehat{\mathit{prot}}_1 \sqsubseteq \mathbf{!}\,\langle v\rangle\{\mathsf{True}\}.\,\mathit{prot}_2\big)\big) &\mathrel{-\!\!*} (\widehat{\mathit{prot}}_1 \sqsubseteq \mathbf{!}\,\vec{x}\!:\!\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot}_2) && (7)\\
\big(\forall \vec{x}\!:\!\vec{\tau}.\,P \mathrel{-\!\!*} \big(\mathbf{?}\,\langle v\rangle\{\mathsf{True}\}.\,\mathit{prot}_1 \sqsubseteq \widehat{\mathit{prot}}_2\big)\big) &\mathrel{-\!\!*} (\mathbf{?}\,\vec{x}\!:\!\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot}_1 \sqsubseteq \widehat{\mathit{prot}}_2) && (8)
\end{aligned}
$$

Session subtyping rules:

$$
\frac{S_1 <: S_2}{\mathtt{chan}\ S_1 <: \mathtt{chan}\ S_2}
\qquad
\frac{A_2 <: A_1 \qquad S_1 <: S_2}{\mathbf{!}A_1.\,S_1 <: \mathbf{!}A_2.\,S_2}
\qquad
\frac{A_1 <: A_2 \qquad S_1 <: S_2}{\mathbf{?}A_1.\,S_1 <: \mathbf{?}A_2.\,S_2}
$$

$$
\begin{aligned}
\mathbf{?}A_1.\,\mathbf{!}A_2.\,S &<: \mathbf{!}A_2.\,\mathbf{?}A_1.\,S\\
\mathbf{!}_{(\vec{X}:\vec{k})}\,A.\,S &<: \mathbf{!}A[\vec{K}/\vec{X}].\,S[\vec{K}/\vec{X}]\\
\mathbf{?}A[\vec{K}/\vec{X}].\,S[\vec{K}/\vec{X}] &<: \mathbf{?}_{(\vec{X}:\vec{k})}\,A.\,S
\end{aligned}
\qquad
\frac{\widehat{S}_1 <: \mathbf{!}A.\,S_2}{\widehat{S}_1 <: \mathbf{!}_{(\vec{X}:\vec{k})}A.\,S_2}
\qquad
\frac{\mathbf{?}A.\,S_1 <: \widehat{S}_2}{\mathbf{?}_{(\vec{X}:\vec{k})}A.\,S_1 <: \widehat{S}_2}
$$

Figure 4 A selection of the Actris 2.0 subprotocol and session subtyping rules.

Subtyping in session types allows sending subtypes and receiving supertypes, as well as increasing and reducing the range of choices for branchings and selections, respectively:

$$
\frac{A_2 <: A_1 \qquad S_1 <: S_2}{\mathbf{!}A_1.\,S_1 <: \mathbf{!}A_2.\,S_2}
\qquad
\frac{A_1 <: A_2 \qquad S_1 <: S_2}{\mathbf{?}A_1.\,S_1 <: \mathbf{?}A_2.\,S_2}
\qquad
\frac{\vec{S}_2 \subseteq \vec{S}_1}{\oplus\{\vec{S}_1\} <: \oplus\{\vec{S}_2\}}
\qquad
\frac{\vec{S}_1 \subseteq \vec{S}_2}{\&\{\vec{S}_1\} <: \&\{\vec{S}_2\}}
$$

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 the context of asynchronous session types, one can further extend subtyping with a "swapping" rule $\mathbf{?}A_1.\,\mathbf{!}A_2.\,S <: \mathbf{!}A_2.\,\mathbf{?}A_1.\,S$ that allows performing sends ahead of receives. This is called *asynchronous subtyping* [34]. For example, using this rule, the client of the "mapper" protocol $\mu\,(\mathit{rec} : \blacklozenge).\,\mathbf{!}_{(X,Y:\bigstar)}\,(X \multimap Y).\,\mathbf{!}X.\,\mathbf{?}Y.\,\mathit{rec}$ from § 3 can "swap" sends ahead of receives. It can thus send multiple requests at once, and only then await the results.

While the original version of Actris was built on top of asynchronous channels, it did not exploit their asynchronous nature since it did not support asynchronous subtyping (in fact, it did not support subtyping at all). We extend Actris to Actris 2.0 by extending it with a *subprotocol relation* $\mathit{prot}_1 \sqsubseteq \mathit{prot}_2$. The subprotocol relation generalises the usual kind of subtyping in session types—including asynchronous subtyping—to the logical level and can be used for functional correctness proofs. In addition, we show that our new subprotocol relation can be used to extend our semantic type system with a rich notion of subtyping. Our notion of subtyping supports new rules for session type polymorphism that integrate with asynchronous subtyping. Lastly, we show that Löb induction can be used to prove subtyping relations for recursive session types.

Subprotocols Actris 2.0 extends Actris with a *subprotocol relation* $\mathit{prot}_1 \sqsubseteq \mathit{prot}_2$ that is a reflexive, transitive, and enjoys the rules in Figure 4. The first four rules generalise the conventional rules of session types: (1) protocol ownership ($c \rightarrowtail \mathit{prot}$) is closed under subprotocols, (2) and (3) subprotocols are compatible with send and receive, (4) subprotocols are closed under "swapping" sends ahead of receives. Rule (5) states that given $\mathbf{!}\, \vec{x}\!:\!\vec{\tau}\, \langle v \rangle \{P\}.\, \mathit{prot}$, we can instantiate the binders $\vec{x}\!:\!\vec{\tau}$ and give up ownership of P before performing the physical send operation. Rule (5) may appear limited, since all binders $\vec{x}\!:\!\vec{\tau}$ and the resource P needs to be instantiated in one go. However, using rule (7) we can first eliminate binders. For example, assuming $\ell_2' \mapsto 22$, this combination of rules makes it possible to prove:

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

Rule (6) and (8) are similar, but for receives. Rule (7) and (8) can only be used if the other protocol is not **end**. Hence, we use the notation $\widehat{\mathit{prot}}$ to refer to non-**end** protocols.

To see subprotocols in action, let us reconsider the example *lockprog* from § 4. Using Actris, we can give this program the following specification for functional correctness:

$$\begin{array}{l}(c \rightarrowtail \mathbf{?}\, (v_1 : \mathsf{Val})\, \langle v_1 \rangle .\mathbf{?}\, (v_2 : \mathsf{Val})\, \langle v_2 \rangle \{\Phi\ v_1\ v_2\}.\, \mathbf{end}) \mathrel{-\!\!*} \\ \mathsf{wp}\ (\mathit{lockprog}\ c)\ \{v.\, \exists (w_1, w_2 : \mathsf{Val}).\, v = (w_1, w_2) * (\Phi\, w_1\, w_2 \vee \Phi\, w_2\, w_1)\}\end{array}$$

This specification states that the program returns a tuple of the two expected integers in any order, satisfying some predicate $\Phi : \mathsf{Val} \to \mathsf{Val} \to \mathsf{iProp}$. With this specification at hand, it would be nonsensical to reprove the program to establish the semantic typing judgement. However, with Actris 2.0's we can prove the following subprotocol relation:

$$\mathtt{?}\mathbf{Z}.\, \mathtt{?}\mathbf{Z}.\, \mathtt{end} \;\sqsubseteq\; \mathbf{?}\, (v_1 : \mathsf{Val})\, \langle v_1 \rangle \{\mathsf{True}\}.\, \mathbf{?}\, (v_2 : \mathsf{Val})\, \langle v_2 \rangle \{\mathbf{Z}\ v_1 * \mathbf{Z}\ v_2\}.\, \mathbf{end}$$

We can then prove $\vDash \mathit{lockprog} : \mathtt{chan}\ (\mathtt{?}\mathbf{Z}.\, \mathtt{?}\mathbf{Z}.\, \mathtt{end}) \multimap (\mathbf{Z} \times \mathbf{Z})$ from rule (1) and the above functional correctness specification by choosing $\Phi\, v_1\, v_2 \triangleq \mathbf{Z}\ w_1 * \mathbf{Z}\ w_2$.

Soundness of Actris 2.0. Similar to the original version, Actris 2.0 is defined as a library in Iris. The type iProto is defined using Iris's solver for recursive domain equations, and the protocol ownership ($c \rightarrowtail \mathit{prot}$) is defined as an Iris definition using its mechanisms for ghost state. Non-trivial modifications to these definitions were needed to add the subprotocol relation ($\mathit{prot}_1 \sqsubseteq \mathit{prot}_2$). The definitions are all mechanised in Coq, but require a level of knowledge about Iris that is out of scope for this paper.

Asynchronous Subtyping Since session types are defined as Actris protocols, we define the semantic subtyping relation for session types as the subprotocol relation:

$$S <: T \triangleq S \sqsubseteq T$$

A selection of subtyping rules is shown in Figure 4. The notation $\widehat{S}$ refers to any non-`end` session type. Due to the close similarity between subtyping and subprotocols, the standard subtyping rules from session types, including the "swapping" rule, are a direct consequence of Actris 2.0's rules. In addition to the standard rules, our type system thus also includes novel rules for dealing with polymorphism in session types via the (asynchronous) subtyping relation. In particular, we can apply the "swapping" rule under polymorphic binders.

Guarded Coinductive Subprotocols Since Actris's protocols and subprotocol relation are defined coinductively (via Iris's machinery for guarded recursion), we can use Iris's support for Löb induction to reason about recursive procotols. Löb induction allows one to assume $\triangleright Q$ ("later" Q) while proving Q. This can, among others, be combined with rules (2) and (3) from Figure 4 for subprotocols. These rules have a later ($\triangleright$) in their premise, and thus make it possible to obtain the Löb induction hypothesis "now" (*i.e.,* without $\triangleright$). For example, Löb induction allows us to prove the following subtyping relation of the "mapper" protocol:

$$\begin{aligned}&\mu\,(\mathit{rec} : \blacklozenge).\, \mathbf{!}_{(X,Y:\bigstar)}\,(X \multimap Y).\, \mathbf{!}X.\, \mathbf{?}Y.\, \mathit{rec}\\ <:\ &\mu\,(\mathit{rec} : \blacklozenge).\, \mathbf{!}_{(X_1,X_2:\bigstar)}\,(X_1 \multimap \mathbf{B}).\, \mathbf{!}X_1.\, \mathbf{!}(X_2 \multimap \mathbf{Z}).\, \mathbf{!}X_2.\, \mathbf{?B}.\, \mathbf{?Z}.\, \mathit{rec}\end{aligned}$$

The idea of proving subtyping coinductively was proposed by Brandt and Henglein [6], but we formulate it in terms of Löb induction, and apply it to session types.

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 [19]. Mostrous *et al.* expanded on this work by adding support for multi-party asynchronous recursive session types [34], and later for higher-order process calculi [33]. Their work also includes an output-input swapping rule, similar to our swapping rule in Figure 4. Even though the original paper claims that subtyping is decidable, it was later proven to not be the case by Bravetti *et al.* [7], precisely because of this swapping rule. Another extension for multi-party session types was done by Padovani *et al.*, who explored adding liveness properties on top of safety [36, 37].

Gay [18] 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.* [14] expanded on this work by adding subtyping and delegation, while still only conjecturing that recursion was a possible extension. Caires *et al.* [8] 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.

Balzer *et al.* [4, 5] 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 for type safety, but also for 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 [31, 25, 21], program refinement [31, 32, 43, 45, 16], robust safety [42], and non-interference [17]. The most immediately related work in this area is the RustBelt project [25], 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.* [21] 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 *et al.* [9], Wadler [46], Carbone *et al.* [10], and Dardha *et al.* [13]. As part of this line of work, Perez *et al.* used logical relations to prove termination (strong normalisation) [38] and confluence [39] 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.* [43] 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.* [22] developed the Actris mechanisation that this work is built on top of. Both lines of work focus on different properties than type safety, and consider neither subprotocols or subtyping.

Gay *et al.* [20] 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. In our setup duality does not distribute over μ, but recursive definitions must be unfolded to expose the session type before duality can be applied. Even so, we can use Löb induction to prove (subtyping) properties of recursive types and their duals.

Castro *et al.* [11] focused on the meta theory 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 [44] 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 book keeping, in particular for properties about resource separation. Rouvoet *et al.* [41] 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.

Craciun *et al.* [12] introduced a separation logic with a session type flavour that has similarities with Actris. Their logic supports mutable state, message passing (including delegation), and branching. Additionally it includes a notion of protocol entailment, which, even though it is similar to subtyping, is not asynchronous and is not a first-class connective of the logic like our notion of subprotocols. The latter means that the types of ownership transfers we allow (rules 5-8 in Figure 4) are not permissible. Their work targets functional correctness rather than type safety, does not support higher-order functions, nor integration with other concurrency primitives such as locks/mutexes. While their work has not been mechanised in a proof assistant, programs can be verified using the HIP/SLEEK verifier.

7 Concluding Remarks and Mechanisation in Coq

We have shown how the foundational semantic approach to type safety can be applied to session typing, and how this results in an extensible *open* type system with many features, including support for manual typing proofs. We conclude this paper by giving an idea on how the semantic approach made the mechanisation in Coq feasible.

By building on top of the Iris and Actris frameworks, we were able to inherit their libraries for various programming constructs, such as locks and channels. These could be integrated into the type system with little effort. By working in the Iris separation logic, we avoid reasoning about explicit resources. Of crucial importance is Iris's support for tactical proofs in Coq [31, 29], which does not just make it possible to reason at the level of separation logic *in theory*, but also *in practice* in Coq. Proofs in Coq are typically in the order of 5-20 lines per semantic typing rule, which is a good indication that our approach is scalable.

Furthermore, by taking the foundational approach to the full extent—*i.e.,* by modelling type formers as combinators on semantic types—we are able to reuse Coq's meta-level variable binding for quantifiers. Our approach thus gives the same feeling of working with higher-order abstract syntax [40], while being semantical instead of syntactical.

References

1 Amal Ahmed. *Semantics of types for mutable state*. PhD thesis, Princeton University, 2004.

2 Amal Ahmed, Andrew W. Appel, Christopher D. Richards, Kedar N. Swadi, Gang Tan, and Daniel C. Wang. Semantic foundations for typed assembly languages. *TOPLAS*, 32(3):7:1–7:67, 2010.

3 Andrew W. Appel and David A. McAllester. An indexed model of recursive types for foundational proof-carrying code. *TOPLAS*, 23(5):657–683, 2001.

4 Stephanie Balzer and Frank Pfenning. Manifest sharing with session types. *PACMPL*, 1(ICFP):37:1–37:29, 2017.

5 Stephanie Balzer, Bernardo Toninho, and Frank Pfenning. Manifest deadlock-freedom for shared session types. In *ESOP*, volume 11423 of *LNCS*, pages 611–639, 2019.

6 Michael Brandt and Fritz Henglein. Coinductive axiomatization of recursive type equality and subtyping. *Fundamenta Informaticae*, 33(4):309–338, 1998.

7 Mario Bravetti, Marco Carbone, and Gianluigi Zavattaro. Undecidability of asynchronous session subtyping. *Information and Computation*, 256:300–320, 2017.

8 Luís Caires, Jorge A. Pérez, Frank Pfenning, and Bernardo Toninho. Behavioral polymorphism and parametricity in session-based communication. In *ESOP*, volume 7792 of *LNCS*, pages 330–349, 2013.

9 Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *CONCUR*, volume 6269 of *LNCS*, pages 222–236, 2010.

10 Marco Carbone, Fabrizio Montesi, Carsten Schürmann, and Nobuko Yoshida. Multiparty session types as coherence proofs. *Acta Informatica*, 54(3):243–269, 2017.

11 David Castro, Francisco Ferreira, and Nobuko Yoshida. EMTST: Engineering the meta-theory of session types. In *TACAS*, volume 12079 of *LNCS*, pages 278–285, 2020.

12 Florin Craciun, Tibor Kiss, and Andreea Costea. Towards a session logic for communication protocols. In *ICECCS*, pages 140–149, 2015.

13 Ornela Dardha and Simon J. Gay. A new linear logic for deadlock-free session-typed processes. In *FOSSACS*, volume 10803 of *LNCS*, pages 91–109, 2018.

14 Ornela Dardha, Elena Giachino, and Davide Sangiorgi. Session types revisited. In *PPDP*, pages 139–150, 2012.

15 Derek Dreyer, Amin Timany, Robbert Krebbers, Lars Birkedal, and Ralf Jung. What type soundness theorem do you really want to prove?, 2019. SIGPLAN blog post, available at `https://blog.sigplan.org/2019/10/17/what-type-soundness-theorem-do-you-really-want-to-prove/`.

16 Dan Frumin, Robbert Krebbers, and Lars Birkedal. ReLoC: A mechanised relational logic for fine-grained concurrency. In *LICS*, pages 442–451, 2018.

17 Dan Frumin, Robbert Krebbers, and Lars Birkedal. Compositional non-interference for fine-grained concurrent programs, 2020. To appear in S&P'21.

18 Simon J. Gay. Bounded polymorphism in session types. *MSCS*, 18(5):895–930, 2008.

19 Simon J. Gay and Malcolm Hole. Subtyping for session types in the pi calculus. *Acta Informatica*, 42(2-3):191–225, 2005.

20 Simon J. Gay, Peter Thiemann, and Vasco T. Vasconcelos. Duality of session types: The final cut. In *PLACES*, volume 314 of *EPTCS*, pages 23–33, 2020.

21 Paolo G. Giarrusso, Léo Stefanesco, Amin Timany, Lars Birkedal, and Robbert Krebbers. Scala step-by-step: Soundness for DOT with step-indexed logical relations in Iris, 2020. Draft.

22 Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. Actris: Session-type based reasoning in separation logic. *PACMPL*, 4(POPL):6:1–6:30, 2020.

23 Jonas Kastberg Hinrichsen, Daniël Louwrink, Jesper Bengtson, and Robbert Krebbers. Coq mechanization of Actris 2.0 and semantic session typing system, 2020. Available online at `https://gitlab.mpi-sws.org/iris/actris/-/tree/concur2020`.

24 Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP*, volume 1381 of *LNCS*, pages 122–138, 1998.

25 Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: Securing the foundations of the rust programming language. *PACMPL*, 2(POPL):66:1–66:34, 2018.

26 Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. Higher-order ghost state. In *ICFP*, pages 256–269, 2016.

27 Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *JFP*, 28:e20, 2018.

28 Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *POPL*, pages 637–650, 2015.

29 Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. MoSeL: A general, extensible modal framework for interactive proofs in separation logic. *PACMPL*, 2(ICFP):77:1–77:30, 2018.

30 Robbert Krebbers, Ralf Jung, Ales Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. The essence of higher-order concurrent separation logic. In *ESOP*, volume 10201 of *LNCS*, pages 696–723, 2017.

31 Robbert Krebbers, Amin Timany, and Lars Birkedal. Interactive proofs in higher-order concurrent separation logic. In *POPL*, pages 205–217, 2017.

32 Morten Krogh-Jespersen, Kasper Svendsen, and Lars Birkedal. A relational model of types-and-effects in higher-order concurrent separation logic. In *POPL*, pages 218–231, 2017.

33 Dimitris Mostrous and Nobuko Yoshida. Session typing and asynchronous subtyping for the higher-order π-calculus. *Information and Computation*, 241:227–263, 2015.

34 Dimitris Mostrous, Nobuko Yoshida, and Kohei Honda. Global principal typing in partially commutative asynchronous sessions. In *ESOP*, volume 5502 of *LNCS*, pages 316–332, 2009.

35 Hiroshi Nakano. A modality for recursion. In *LICS*, pages 255–266, 2000.

36 Luca Padovani. Fair subtyping for multi-party session types. In *COORDINATION*, volume 6721 of *LNCS*, pages 127–141, 2011.

37 Luca Padovani. Fair subtyping for multi-party session types. *MSCS*, 26(3):424–464, 2016.

38 Jorge A. Pérez, Luís Caires, Frank Pfenning, and Bernardo Toninho. Linear logical relations for session-based concurrency. In *ESOP*, volume 7211 of *LNCS*, pages 539–558, 2012.

39 Jorge A. Pérez, Luís Caires, Frank Pfenning, and Bernardo Toninho. Linear logical relations and observational equivalences for session-based concurrency. *Information and Computation*, 239:254–302, 2014.

40 Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *PLDI*, pages 199–208, 1988.

41 Arjen Rouvoet, Casper Bach Poulsen, Robbert Krebbers, and Eelco Visser. Intrinsically-typed definitional interpreters for linear, session-typed languages. In *CPP*, pages 284–298. ACM, 2020.

42 David Swasey, Deepak Garg, and Derek Dreyer. Robust and compositional verification of object capability patterns. *PACMPL*, 1(OOPSLA):89:1–89:26, 2017.

43 Joseph Tassarotti, Ralf Jung, and Robert Harper. A higher-order logic for concurrent termination-preserving refinement. In *ESOP*, volume 10201 of *LNCS*, pages 909–936, 2017.

44 Peter Thiemann. Intrinsically-typed mechanized semantics for session types. In *PPDP*, pages 19:1–19:15, 2019.

45 Amin Timany, Léo Stefanesco, Morten Krogh-Jespersen, and Lars Birkedal. A logical relation for monadic encapsulation of state: Proving contextual equivalences in the presence of runST. *PACMPL*, 2(POPL):64:1–64:28, 2018.

46 Philip Wadler. Propositions as sessions. In *ICFP*, pages 273–286, 2012.