Machine-Checked Semantic Session Typing

Jonas Kastberg Hinrichsen
IT University of Copenhagen, Denmark

Daniël Louwrink
University of Amsterdam, The Netherlands

Robbert Krebbers
Radboud University and 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 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 the typing judgements of racy, but safe, programs that cannot be type checked using only the rules of the type system.

1 Introduction

Session types [Honda et al. 1998] 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 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 embracing the *semantic approach* to type safety [Ahmed 2004; Ahmed et al. 2010; Appel and McAllester 2001], 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 Iris [Jung et al. 2016, 2018b, 2015; Krebbers et al. 2018, 2017a,b] giving us a high degree of trust that they are correct.

The syntactic approach 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 is that it does not require progress and preservation proofs 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:

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

Two values are requested over channel c in parallel, and returned as a tuple (using the operator || for parallel composition, and the type $\mathsf{chan}\ (?\mathbf{Z}.\, ?\mathbf{Z}.\, \mathsf{end})$ 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 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

[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.

the semantic approach, we can prove the typing lemma $\vDash \lambda\, c.\, (\texttt{recv}\ c \mid\mid \texttt{recv}\ c) : \mathsf{chan}\ (?\mathbf{Z}.\, ?\mathbf{Z}.\, \mathsf{end}) \multimap (\mathbf{Z} \times \mathbf{Z})$ by appealing to 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. 2016, 2018b, 2015; Krebbers et al. 2017a] has proved to be sufficiently expressive to define semantic type systems for *e.g.,* Rust [Jung et al. 2018a, 2020] 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 use of the Actris framework for message-passing in Iris [Hinrichsen et al. 2020a,b]. Actris includes the notion of *dependent separation protocols*, which are similar to 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. 2018, 2017b].

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). 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 (§ 3).
- We demonstrate the benefit of our type system being semantic by integrating the manual verification of safe but not conventionally type-checkable programs (§ 4).
- We provide insight on the benefits of a semantic type system in regards to mechanisation efforts (§ 5). All of our results are mechanised in Coq and can be found in [Anonymous Authors 2020].

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 (§ 2.1). We provide a brief overview of Iris (§ 2.2), and then present a lightweight affine type system (§ 2.3) as the core upon which we built our session-type system (§ 2.4). Our affine type system is inspired by RustBelt [Jung et al. 2018a, 2020], but drops Rust-specific features like borrowing and lifetimes to focus on session types.

2.1 Language

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

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

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. 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-set.

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 expressive 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 $(P \land Q)$, implication $(P \Rightarrow Q)$, universal $(\forall x : \tau.\, P)$ and existential $(\exists x : \tau.\, P)$ quantification, as well as the connectives of separation logic:

- The *points-to connective* $(\ell \mapsto v)$ asserts 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 it enjoys 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.

Term types:

$$\begin{aligned}
\mathsf{Type}_{\bigstar} &\triangleq \mathsf{Val} \rightarrow \mathsf{iProp} \\
\mathsf{any} &\triangleq \lambda\, w.\, \mathsf{True} \\
\mathbf{1} &\triangleq \lambda\, w.\, w \in \{()\} \\
\mathbf{B} &\triangleq \lambda\, w.\, w \in \mathbb{B} \\
\mathbf{Z} &\triangleq \lambda\, w.\, w \in \mathbb{Z} \\
\mathtt{ref}_{\mathtt{uniq}}\, A &\triangleq \lambda\, w.\, \exists v.\, w \in \mathsf{Loc} * (w \mapsto v) * \triangleright(A\, v) \\
A_1 \times A_2 &\triangleq \lambda\, w.\, \exists v_1, v_2.\, w = (v_1, v_2)\ * \triangleright(A_1\, v_1) * \triangleright(A_2\, v_2) \\
A_1 + A_2 &\triangleq \lambda\, w.\, \exists v.\, (w = \mathtt{inl}\ v * \triangleright(A_1\, v)) \lor (w = \mathtt{inr}\ v * \triangleright(A_2\, v)) \\
A \multimap B &\triangleq \lambda\, w.\, \forall v.\ \triangleright(A\, v) \mathrel{-\!\!*} \mathsf{wp}\ (w\ v)\ \{B\} \\
\mathsf{chan}\, S &\triangleq \lambda\, w.\, w \rightarrowtail S
\end{aligned}$$

Judgements:

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

Session types:

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

Figure 1. Typing judgements and type formers of the semantic type system.

As we see in §2.3 these connectives match up with the type formers for unique references, products, and affine functions.

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 a connective ($c \rightarrowtail \mathit{prot}$) that asserts resource ownership of a channel, to support reasoning about functional correctness of message-passing programs (§ 2.4), and in this paper to semantically type safe yet not conventionally type-checkable programs (§ 4).

To define recursive types semantically (§ 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 later ($\triangleright$) can be eliminated by taking a program step, which is formalised by Iris's proof rule $\triangleright P * \mathsf{wp}\ e\ \{\Phi\}$ entails $\mathsf{wp}\ e\ \{w.\ P * \Phi\, w\}$ if $e \notin \mathsf{Val}$. This means that $\triangleright P$ can also be read as "P holds after one more step of computation."

In this paper we will not further detail the semantics of Iris, but refer the interested reader to Jung et al. [2018b].

2.3 Term Types

The definitions of our semantic type system 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 $\mathsf{Type}_{\bigstar}$ are defined as Iris predicates over values, and session types $\mathsf{Type}_{\blacklozenge}$ are defined as dependent separation protocols of Actris (§ 2.4).

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 $\mathtt{ref}_{\mathtt{uniq}}\, A$ 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 ($\mathrel{-\!\!*}$) 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 $w\, v$.

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 § 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 1) with a *pre-* and *post-context* $\Gamma, \Gamma' \in \mathsf{List}(\mathsf{String} \times \mathsf{Type}_{\bigstar})$, 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 auxiliary judgement employs the iterated separation conjunction $\mathop{\Large\ast}_{(x,A)\in\Gamma}$ 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 $\Gamma \vDash e : A \dashv \Gamma'$ 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 postcondition 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'.

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

Selection of Iris's proof rules for weakest preconditions:

$$\Phi\, v \mathrel{-\!\!*} \mathsf{wp}\; v\; \{\Phi\} \tag{wp-val}$$

$$\ell \mapsto v \mathrel{-\!\!*} \mathsf{wp}\; !\ell\; \{w.\, (v = w) * (\ell \mapsto v)\} \tag{wp-load}$$

$$\mathsf{wp}\; e_1 \left\{v.\, \mathsf{wp}\; e_2[v/x]\; \{\Phi\}\right\} \mathrel{-\!\!*} \mathsf{wp}\; (\mathtt{let}\, x = e_1\, \mathtt{in}\, e_2)\; \{\Phi\} \tag{wp-let}$$

$$\mathsf{wp}\; e_1\; \{\Phi_1\} * \mathsf{wp}\; e_2\; \{\Phi_2\} \mathrel{-\!\!*} \mathsf{wp}\; (e_1 \mid\mid e_2)\; \{v.\, \exists v_1, v_2.\, (v = (v_1, v_2)) * \Phi_1\, v_1 * \Phi_2\, v_2\} \tag{wp-par}$$

Selection of semantic typing rules:

$$\Gamma \vDash i : \mathbf{Z} \qquad \Gamma, (x : A) \vDash x : A \dashv \Gamma, (x : \mathsf{any}) \qquad \Gamma, (x : \mathsf{ref}_{\mathsf{uniq}}\, A) \vDash\, !x : A \dashv \Gamma, (x : \mathsf{ref}_{\mathsf{uniq}}\, \mathsf{any})$$

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

Figure 2. A selection of Iris's proof rules and semantic typing rules.

definition of the semantic typing judgement $\Gamma \vDash e : A \dashv \Gamma'$, and proving the corresponding proposition in Iris using Iris's rules for weakest preconditions. A selection of typing rules, along with Iris's weakest precondition rules used to prove them, is presented in Figure 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 $\Gamma, (x : \mathsf{any})$ as ownership of A has been moved out. For substructural type systems this is crucial as in expressions such as $\mathtt{let}\, x = y\, \mathtt{in}\, e$, 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 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 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 [Jung et al. 2018b; Krebbers et al. 2017a]. 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.

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 $\vec{S} : \mathbb{Z} \rightharpoonup \mathsf{Type}_{\blacklozenge}$, 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. 2020b], 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 $!\,\vec{x} : \vec{\tau}\, \langle v \rangle \{P\}.\, \mathit{prot}$ and $?\,\vec{x} : \vec{\tau}\, \langle v \rangle \{P\}.\, \mathit{prot}$ 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 $!\,(\ell : \mathsf{Loc})\; (i : \mathbb{Z})\; \langle \ell \rangle \{\ell \mapsto i * 10 \leq i\}.\; ?\, \langle () \rangle \{\ell \mapsto (i + 1)\}.\, \mathbf{end}$ 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 $c \rightarrowtail \mathit{prot}$ denotes ownership of a channel c with a dependent separation protocol prot. The Actris proof rules are shown in Figure 3. The rule for $\mathtt{new\_chan}\; ()$ allows ascribing any protocol to a new 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 (?) is turned into a send (!), and *vice versa.* The rule for $\mathtt{send}\; c\; w$ 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 $c \rightarrowtail\, !\,\vec{x} : \vec{\tau}\, \langle v \rangle \{P\}.\, \mathit{prot}$, 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 $c \rightarrowtail \mathit{prot}[\vec{t}/\vec{x}]$. The rule for $\mathtt{recv}\; c$ is essentially dual to the rule for $\mathtt{send}\; c\; w$. One needs to give up ownership of $c \rightarrowtail\, ?\,\vec{x} : \vec{\tau}\, \langle v \rangle \{P\}.\, \mathit{prot}$, and in return acquires the resources $P[\vec{y}/\vec{x}]$, the return value w where $w = v[\vec{y}/\vec{x}]$, and finally

Actris's proof rules for dependent separation protocols:

$$\mathsf{wp}\ \texttt{new\_chan}\ ()\ \big\{v.\ \exists c, c'.\ (v = (c, c')) * c \rightarrowtail \mathit{prot} * c' \rightarrowtail \overline{\mathit{prot}}\big\} \qquad (\textsc{wp-newchan})$$

$$c \rightarrowtail {!}\,\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot} * \triangleright P[\vec{t}/\vec{x}] \mathrel{-\!\!*} \mathsf{wp}\ \texttt{send}\ c\ (v[\vec{t}/\vec{x}])\ \big\{c \rightarrowtail \mathit{prot}[\vec{t}/\vec{x}]\big\} \qquad (\textsc{wp-send})$$

$$c \rightarrowtail {?}\,\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot} \mathrel{-\!\!*} \mathsf{wp}\ \texttt{recv}\ c\ \{w.\ \exists\vec{y}.\ (w = v[\vec{y}/\vec{x}]) * c \rightarrowtail \mathit{prot}[\vec{y}/\vec{x}] * P[\vec{y}/\vec{x}]\} \qquad (\textsc{wp-recv})$$

Semantic typing rules for channels:

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

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

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

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

Figure 3. Actris's proof rules for dependent separation protocols and semantic typing rules for channels.

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 $\mathsf{chan}\ S$ is defined in terms of Actris's connective for channel ownership $w \rightarrowtail S$. The definition of the terminator (end), send (!), and receive (?) follow from their dependent separation protocol counterparts. For example ${!}A.\,S$ is defined as ${!}\,(v : \mathsf{Val})\,\langle v\rangle\{A\ v\}.\,S$. It says that a value v is sent along with ownership of $A\ v$.

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 ${!}\,(l : \mathbb{Z})\,\langle l\rangle\big\{l \in \mathrm{dom}(\vec{S})\big\}.\,\vec{S}(l)$. 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.

The duality $\overline{S}$ of session types S is inherited from Actris, and our encoding of branch (&) and select (⊕) ensures that the dual of a session type turns all sends and selects respectively into receives and branches, and *vice versa.*

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

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

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

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

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 § 2.3 remains applicable without change.

3 Extending the Type System

We demonstrate the extensibility of our approach to session types by adding term- and session-level subtyping (§ 3.1 and § 3.6), copyable types (§ 3.2), term- and session-level equi-recursive types (§ 3.3), term- and session-level polymorphism (§ 3.4), and locks/mutexes (§ 3.5). While we only present a small representative selection of rules associated with each extension, all rules can be found in Appendix A.

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 together with the conventional subtyping rules as lemmas. For example:

$$\frac{\Gamma_1 <:_{\mathbf{ctx}} \Gamma_1' \qquad \Gamma_1' \vDash e : A \dashv \Gamma_2' \qquad A <: B \qquad \Gamma_2' <:_{\mathbf{ctx}} \Gamma_2}{\Gamma_1 \vDash e : B \dashv \Gamma_2}$$

$$A <: \mathsf{any} \qquad A <: A \qquad \frac{A <: B \qquad B <: C}{A <: C}$$

The proof of the subsumption rule make use of the 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. We will see more interesting subtyping rules in § 3.2 and § 3.6.

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 § 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 $\mathtt{copy}$ and a property $\mathtt{copyable}$:

$$\mathtt{copy}\, A \triangleq \lambda\, w.\ \Box(A\, w) \qquad \mathtt{copyable}\, A \triangleq A <: \mathtt{copy}\, A$$

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 $A \in \left\{\mathtt{ref}_{\mathtt{uniq}}\, B, \mathsf{chan}\, S\right\}$ 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 $A \to B \triangleq \mathtt{copy}\, (A \multimap B)$ as the type of *unrestricted* functions, that can be applied any number of times.

The type former $\mathtt{copy}$ 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 $\Box\, P \mathrel{-\!\!*} (\Box\, P * P)$. This allows copyable types occurring in the context to be duplicated:

$$(x : A) <:_{\mathbf{ctx}} (x : A), (x : A) \qquad \text{if } \mathtt{copyable}\, A$$

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, 2020] 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 $\mathtt{copy}$ type former and a subtyping rule.

3.3 Equi-Recursive Term and Session Types

We extend the type system with equi-recursive types using Iris's fixpoint operator. Recall from § 2.2 that Iris's fixpoint operator requires that recursive definitions are contractive, meaning that recursive occurrences appear below a later ($\triangleright$). A recursive occurrence is also considered guarded when it appears in (1) the postcondition Φ of a weakest precondition $\mathsf{wp}\ e\ \{\Phi\}$ with $e \notin \mathsf{Val}$, (2) the tail *prot* of a protocol $\mathbf{!}\, \vec{x} : \vec{\tau}\, \langle v \rangle \{P\}.\, \mathit{prot}$ or $\mathbf{?}\, \vec{x} : \vec{\tau}\, \langle v \rangle \{P\}.\, \mathit{prot}$, and (3) the protocol *prot* of a channel ownership $c \rightarrowtail \mathit{prot}$, as these constructs contain $\triangleright$ modalities internally.

We lift the guarded recursion operator of Iris into a *kinded* operator for equi-recursion in the type system:

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

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 $\mu\, (X : \blacklozenge).\, \mathbf{!}(\mathsf{chan}\, X).\, X$, 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).

3.4 Polymorphism in Term and Session 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 $\forall (X : \blacklozenge).\, \mathsf{chan}\ (\mathbf{!}\mathbb{Z}.X) \multimap \mathsf{chan}\, X$ (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).

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: that is, the inhabitants of universal types are all *thunks*. By using explicit thunks, we avoid having to impose an ML-like *value restriction* [Wright 1995] to ensure type soundness in the presence of imperative side-effects.

The typing rules for term-level polymorphism are standard and can be found in Appendix A.

A more interesting extension is polymorphism in session types [Gay 2008]. An example is the following type describing the interaction with a polymorphic computation service:

$$\begin{aligned}&\texttt{compute\_type} \triangleq \mu\, (\mathit{rec} : \blacklozenge).\\ &\qquad \oplus\{\texttt{cont} : \mathbf{!}_{(X:\bigstar)}\ (\mathbf{1} \multimap X).\, \mathbf{?}X.\, \mathit{rec}, \texttt{stop} : \texttt{end}\}\end{aligned}$$

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:

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

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 (*i.e.,* they allow quantification over Iris predicates). 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:

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

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.

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
    cont ⇒ let f = recv c in send (f ()) ; go c
  | stop ⇒ ()
  end
```

We show $\vDash \texttt{compute\_service} : \mathsf{chan}\ \overline{\texttt{compute\_type}} \to ()$ using only the type inference rules of our semantic type system. In §4.2 we show a client that uses this service but which cannot itself be type checked using our inference rules but rather requires a manual proof of its typing judgement.

3.5 Locks and Mutexes

The substructural nature of channels (of type $\mathsf{chan}\ S$) 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 $\texttt{mutex}\, A$ 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 computation service `compute_type` from §3.4 between multiple clients—they can acquire the mutex ($\texttt{mutex}\ \texttt{compute\_type}$), 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 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 `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:

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

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 and make use of the representation predicate $\texttt{isLock}\ \mathit{lk}\ R$, 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 $\texttt{isLock}\ \mathit{lk}\ R$. The representation is persistent, so it can be freely duplicated. When entering a critical section using $\texttt{acquire}\ \ \mathit{lk}$, a thread gets exclusive ownership of R, which has to be given up when releasing the lock using $\texttt{release}\ \ \mathit{lk}$. Using the lock representation predicate, we define type formers for mutexes:

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

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

Iris's proof rules for locks:

$$\texttt{isLock}\ \mathit{lk}\ R \mathrel{-\!\!*} \Box(\texttt{isLock}\ \mathit{lk}\ R)$$

$$R \mathrel{-\!\!*} \mathsf{wp}\ \texttt{newlock}\ ()\ \{\mathit{lk}.\ \texttt{isLock}\ \mathit{lk}\ R\}$$

$$\texttt{isLock}\ \mathit{lk}\ R \mathrel{-\!\!*} \mathsf{wp}\ \texttt{acquire}\ \mathit{lk}\ \{R\}$$

$$\texttt{isLock}\ \mathit{lk}\ R * R \mathrel{-\!\!*} \mathsf{wp}\ \texttt{release}\ \mathit{lk}\ \{\mathsf{True}\}$$

Semantic typing rules for mutexes:

$$\texttt{copyable}\,(\texttt{mutex}\,A) \qquad \Gamma, (x : A) \vDash \texttt{newmutex}\ x : \texttt{mutex}\,A \dashv \Gamma$$

$$\Gamma, (x : \texttt{mutex}\,A) \vDash \texttt{acquiremutex}\ x : A \dashv \Gamma, (x : \overline{\texttt{mutex}}\,A)$$

$$\frac{\Gamma \vDash e : A \dashv \Gamma', (x : \overline{\texttt{mutex}}\,A)}{\Gamma \vDash \texttt{releasemutex}\ x\ e : \mathbf{1} \dashv \Gamma', (x : \texttt{mutex}\,A)}$$

Figure 4. Iris's proof rules for locks and semantic typing rules for mutexes.

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 are proven as lemmas.

3.6 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:

$$\frac{S <: T}{\mathsf{chan}\ S <: \mathsf{chan}\ T}$$

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:

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

$$\frac{\vec{S}_2 \subseteq \vec{S}_1}{\oplus\{\vec{S}_1\} <: \oplus\{\vec{S}_2\}} \qquad \frac{\vec{S}_1 \subseteq \vec{S}_2}{\&\{\vec{S}_1\} <: \&\{\vec{S}_2\}}$$

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 $\mathbf{?}A_1.\,\mathbf{!}A_2.\,S <: \mathbf{!}A_2.\,\mathbf{?}A_1.\,S$ that allows performing sends (**!**) ahead of receives (**?**), and similar rules that allow performing selects (⊕) ahead of receives (**?**), sends (**!**) ahead of branches (&), and selects (⊕) ahead of branches (&) [Mostrous et al. 2009][2]. For example, using swapping, a client of the computation service from §3.4, 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 results.

To extend our semantic session types with subtyping, we make use of Actris subprotocols [Hinrichsen et al. 2020a], for which the rules are shown in Figure 10. 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) eliminate binders and propositions of right-hand side sending protocols (sp-send-elim) and (2) left-hand side receiving protocols (sp-recv-elim), and (3) instantiate binders of right-hand side sending protocols (sp-send-intro) and (4) left-hand side receiving protocols (sp-recv-intro). Rule sp-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 sp-send-mono and sp-recv-mono account for the monotonicity of the subprotocol relation in the tails, and rule sp-chan-mono states that Actris's connective for channel ownership is closed under the subprotocol relation. Finally, 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 10 for Actris's subprotocol relation. These 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 10. 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.

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.

[2] Discrepancies in the direction between the swapping rules of Mostrous et al. [2009] and us will be discussed in §6.

Actris's proof rules for subprotocols:

$$\begin{array}{rcll}
\big(\forall \vec{x}{:}\vec{\tau}.\, P \mathrel{-\!\!*} \big(prot_1 \sqsubseteq \mathbf{!}\,\langle v\rangle.prot_2\big)\big) & \mathrel{-\!\!*} & (prot_1 \sqsubseteq \mathbf{!}\,\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\,prot_2) \quad \text{if each } \tau \in \vec{\tau} \text{ is inhabited} & (\textsc{sp-send-elim}) \\
\big(\forall \vec{x}{:}\vec{\tau}.\, P \mathrel{-\!\!*} (\mathbf{?}\,\langle v\rangle.prot_1 \sqsubseteq prot_2)\big) & \mathrel{-\!\!*} & (\mathbf{?}\,\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\,prot_1 \sqsubseteq prot_2) \quad \text{if each } \tau \in \vec{\tau} \text{ is inhabited} & (\textsc{sp-recv-elim}) \\
P[\vec{t}/\vec{x}] \mathrel{-\!\!*} (\mathbf{!}\,\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\,prot & \sqsubseteq & \mathbf{!}\,\langle v[\vec{t}/\vec{x}]\rangle.prot[\vec{t}/\vec{x}]) & (\textsc{sp-send-intro}) \\
P[\vec{t}/\vec{x}] \mathrel{-\!\!*} (\mathbf{?}\,\langle v[\vec{t}/\vec{x}]\rangle.prot[\vec{t}/\vec{x}] & \sqsubseteq & \mathbf{?}\,\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\,prot) & (\textsc{sp-recv-intro}) \\
\mathbf{?}\,\langle v_1\rangle\{P_1\}.\,\mathbf{!}\,\langle v_2\rangle\{P_2\}.\,prot & \sqsubseteq & \mathbf{!}\,\langle v_2\rangle\{P_2\}.\,\mathbf{?}\,\langle v_1\rangle\{P_1\}.\,prot & (\textsc{sp-swap}) \\
\triangleright(prot_1 \sqsubseteq prot_2) & \mathrel{-\!\!*} & (\mathbf{!}\,\langle v\rangle\{P\}.\,prot_1 \sqsubseteq \mathbf{!}\,\langle v\rangle\{P\}.\,prot_2) & (\textsc{sp-send-mono}) \\
\triangleright(prot_1 \sqsubseteq prot_2) & \mathrel{-\!\!*} & (\mathbf{?}\,\langle v\rangle\{P\}.\,prot_1 \sqsubseteq \mathbf{?}\,\langle v\rangle\{P\}.\,prot_2) & (\textsc{sp-recv-mono}) \\
(prot_1 \sqsubseteq prot_2) \mathrel{-\!\!*} (c \rightarrowtail prot_1) & \mathrel{-\!\!*} & (c \rightarrowtail prot_2) & (\textsc{sp-chan-mono})
\end{array}$$

Semantic subtyping rules for session polymorphism:

$$\frac{S_1 <: \mathbf{!}A.\,S_2}{S_1 <: \mathbf{!}_{(\vec{X}:\vec{k})}A.\,S_2} \qquad \frac{\mathbf{?}A.\,S_1 <: S_2}{\mathbf{?}_{(\vec{X}:\vec{k})}A.\,S_1 <: S_2} \qquad \mathbf{!}_{(\vec{X}:\vec{k})}\,A.\,S <: \mathbf{!}A[\vec{K}/\vec{X}].\,S[\vec{K}/\vec{X}] \qquad \mathbf{?}A[\vec{K}/\vec{X}].\,S[\vec{K}/\vec{X}] <: \mathbf{?}_{(\vec{X}:\vec{k})}\,A.\,S$$

Figure 5. A selection of the Actris's proof rules for subprotocols and semantic session subtyping rules.

[2018a, 2020], 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 (§4.1), and then show a more realistic example by proving the typing judgement of a parallel client of the computation service from §3.4 that uses a producer/consumer pattern (§4.2).

4.1 Receiving in Parallel

Consider the example from the introduction (where the locks have been made explicit):

```
threadprog ≜ λ c lk. acquire lk; let x = recv c in
                     release lk; x
  lockprog ≜ λ c. let lk = newlock () in
                  (threadprog c lk || threadprog c lk)
```

We want to prove $\vDash$ `lockprog` : chan (**?**Z. **?**Z. end) $\multimap$ (**Z** × **Z**). 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:

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

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:

$$\begin{array}{lll}
\mathtt{chaninv} \triangleq & & \\
\quad (c \rightarrowtail \mathbf{?}\,(v_1 : \mathsf{Val})\,\langle v_1\rangle\{v_1 \in \mathbb{Z}\}. & & \\
\qquad \mathbf{?}\,(v_2 : \mathsf{Val})\,\langle v_2\rangle\{v_2 \in \mathbb{Z}\}.\,\mathbf{end}) & \vee & (i) \\
\quad (c \rightarrowtail \mathbf{?}\,(v_2 : \mathsf{Val})\,\langle v_2\rangle\{v_2 \in \mathbb{Z}\}.\,\mathbf{end}) \ast \boxed{1/2}^{\gamma} & \vee & (ii) \\
\quad (c \rightarrowtail \mathbf{end}) \ast \boxed{1}^{\gamma} & & (iii)
\end{array}$$

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 `isLock` *lk* `chaninv` is allocated by giving up ownership of the channel *c*, where `chaninv` 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 `isLock` *lk* `chaninv`. Both threads have the same proof obligation

$$\mathtt{isLock}\ lk\ \mathtt{chaninv} \ast \boxed{1/2}^{\gamma} \mathrel{-\!\!*} \mathtt{wp}\ \mathtt{threadprog}\ c\ lk\ \{v.\, v \in \mathbb{Z}\}$$

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).

4.2 A Parallel Computation Client

In §3.4 we considered the session type `compute_type` for a client of a polymorphic recursive computation service. We now consider a client `compute_client`, shown in Figure 6, that interacts with the service by sending a list of computation requests and receiving their results in parallel, similar to the producer-consumer patterns.[3] We want to prove:

$$\begin{aligned} \vDash \mathtt{compute\_client} : \mathtt{list}\ (\mathbf{1} \multimap A) \multimap \\ \mathtt{chan}\ \mathtt{compute\_type} \multimap \mathtt{list}\ A \end{aligned}$$

where $\mathtt{list}\ A \triangleq \mu\, rec.\ \mathtt{ref}_{\mathtt{uniq}}\ (\mathbf{1} + (A \times rec))$.

The producer `send_all` and consumer `recv_all` race for a shared lock lk, to send computations and receive results on the shared channel endpoint c (the computation service has the other endpoint), for each element in the linked list input l. They respectively increment and decrement a shared counter $cntr$, to keep track of how many requests that are being processed. The computation results are stored in a new list l', which is returned once `send_all` and `recv_all` terminate. The type system cannot type check such a program, as (1) its behaviour 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 type check the program `compute_client`, we unfold its typing judgement, and resolve each step of the program in sequence, by applying the related weakest precondition rules. We first use the weakest precondition rule for `llength`, found in Figure 6, to convert the type predicate $\mathtt{list}\ (\mathbf{1} \multimap A)$ of the list reference l into the separation-logic list representation predicate $l \overset{\text{list}}{\mapsto}_{(1 \multimap A)} \vec{v}$, which additionally makes the contents of the list $\vec{v}$ explicit. This predicate is defined as:

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

The remainder of the proof then follows from reasoning similar to that of the parallel receive, using a fractional permission $\boxed{1}^{\gamma}$ along with the shared counter $cntr$ to determine the state of the shared channel. To share the counter and the channel, they are put into the lock invariant `chaninv`:

$$\begin{aligned} \mathtt{chaninv} \triangleq \exists n.\ cntr \mapsto n \ast \\ (c \rightarrowtail ((?A)^n \cdot \mathtt{compute\_type}) \vee \quad & (i) \\ (c \rightarrowtail ((?A)^n \cdot \mathtt{end}) \ast \boxed{1}^{\gamma})) \quad & (ii) \end{aligned}$$

The invariant states that the session type of the channel starts with a sequence of receive actions $(?A)^n$, where n is the value of the shared counter $cntr$. 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 has 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 $\boxed{1}^{\gamma}$, after which the weakest precondition rules for `send_all`, `recv_all`, and parallel composition are used. To close the proof, we weaken the list reference returned by `recv_all` to the list reference type `list` A, by forgetting its explicit values $\vec{w}$. The proof of the weakest precondition rule for `send_all` follow as, (1) owning $\boxed{1}^{\gamma}$ means the channel is open so, (2) it can be unfolded, after which the select and send action can be swapped ahead of any arbitrary number of receives, after which (3) incrementing the shared counter $cntr$ lets us close the invariant, and finally (4) we can close the channel by giving up $\boxed{1}^{\gamma}$. The proof of the weakest precondition rule for `recv_all` follow as, (1) we only receive when the shared counter $cntr$ is non-zero so, (2) the head of the channel type will be a receive, and finally (3) decrementing the shared counter $cntr$ lets us close the invariant.

5 Mechanisation in Coq

In this paper, we have used what is often called the "foundational approach" to semantic type safety [Ahmed 2004; Ahmed et al. 2010; Appel and McAllester 2001]. 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 Iris Proof Mode, which provides tactics tailored for reasoning about the connectives of separation logic, and thereby hides unnecessary details related to the embedding of separation logic [Krebbers et al. 2018, 2017b].

Coq Definitions. Term and session types are represented as a dependent type indexed by a kind:[4]

[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.

[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.

```
compute_client ≜ λ l c.
  let n = llength l in
  let cntr = ref 0 in
  let l′ = lnil () in
  let lk = newlock () in
  (send_all l cntr lk c ||
   recv_all l′ n cntr lk c);
  l′
```

```
send_all ≜ rec go l cntr lk c =
  if lisnil l then
    acquire lk; select c stop;
    release lk; ()
  else
    acquire lk;
      select c cont; send c (lpop l);
      cntr ← ! cntr + 1
    release lk; go l cntr lk c
```

```
recv_all  ≜ rec go l n cntr lk c =
  if n = 0 then () else
  acquire lk;
  if ! cntr = 0 then
    release lk; go l n cntr lk c
  else
    let x = recv c in
    cntr ← ! cntr − 1;
    release lk; go l (n − 1) cntr lk c;
    lcons x l
```

$$\mathtt{list}\ A\ l \mathrel{-\!\!*} \mathtt{wp}\ \mathtt{llength}\ l \left\{n.\ n = |\vec{v}| * l \overset{\text{list}}{\mapsto}_A \vec{v}\right\}$$

$$\mathtt{isLock}\ lk\ \mathtt{chaninv} * \boxed{1}^{\gamma} * l \overset{\text{list}}{\mapsto}_{(\mathbf{1} \multimap A)} \vec{v} \mathrel{-\!\!*} \mathtt{wp}\ \mathtt{send\_all}\ l\ cntr\ lk\ c \left\{l \overset{\text{list}}{\mapsto}_{(\mathbf{1} \multimap A)} [\,]\right\}$$

$$\mathtt{isLock}\ lk\ \mathtt{chaninv} * l \overset{\text{list}}{\mapsto}_A [\,] \mathrel{-\!\!*} \mathtt{wp}\ \mathtt{recv\_all}\ l\ n\ cntr\ lk\ c \left\{\exists \vec{w}.\ |\vec{w}| = n * l \overset{\text{list}}{\mapsto}_A \vec{w}\right\}$$

Figure 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).

```
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).
```

The semantic term typing judgement is defined as:

```
Inductive ctx_item Σ :=
  CtxItem { ctx_item_name: string; ctx_item_type: ltty Σ }.
Notation ctx Σ := (list (ctx_item Σ)).

(* 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 which is extended by the Actris framework with connectives for message passing. HeapLang, and thus our typing contexts `ctx`, use strings for variables. Compared to *e.g.*, De Bruijn indices or locally nameless, this makes it possible to write programs in a human-readable way.[5]

The typing judgement is identical to the definition in §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 additional flexibility in manual typing proofs, *e.g.*, it makes it possible to prove typing judgements using Löb induction, without having to unfold the definition. To ensure that the typing judgement can be used as a 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 that 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**.

```
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 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).
```

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. 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

[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.

`∀ κ` in the premise implicitly ensures that `κ` is fresh. The proof of this typing rule is 4 lines of code.

The session type for selection (⊕) 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)).

Lemma ltyped_select Γ (x: string) (i: Z) S Ss :
  Γ !! x = Some (chan (lty_select Ss)) →
  Ss !! i = Some S →
  Γ ⊨ select x #i : () ⫤ env_cons x (chan S) Γ.
```

Since ⊕ 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 $\mathbf{!}\,\vec{x} : \vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot}$ or $\mathbf{?}\,\vec{x} : \vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot}$ 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 makes use of Actris's proof rules, and is 6 lines.

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 § 3.6, 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]; 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 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 [Giarrusso et al. 2020; Jung et al. 2018a; Krebbers et al. 2017b], program refinement [Frumin et al. 2018; Krebbers et al. 2017b; Krogh-Jespersen et al. 2017; Tassarotti et al. 2017; Timany 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 (strong normalisation) [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. [2020a,b] 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. 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 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 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 [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 book keeping, 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.

References

Amal Ahmed. 2004. *Semantics of types for mutable state.* Ph.D. Dissertation. Princeton University.

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), 7:1–7:67.

Anonymous Authors. 2020. Coq Mechanization of "Machine-Checked Semantic Session Typing". Available as anonymous supplementary material in HotCRP.

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

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

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

Stephanie Balzer, Bernardo Toninho, and Frank Pfenning. 2019. Manifest Deadlock-Freedom for Shared Session Types. In *ESOP (LNCS, Vol. 11423)*. 611–639.

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

Luís Caires, Jorge A. Pérez, Frank Pfenning, and Bernardo Toninho. 2013. Behavioral Polymorphism and Parametricity in Session-Based Communication. In *ESOP (LNCS, Vol. 7792)*. 330–349.

Luís Caires and Frank Pfenning. 2010. Session Types as Intuitionistic Linear Propositions. In *CONCUR (LNCS, Vol. 6269)*. 222–236.

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

David Castro, Francisco Ferreira, and Nobuko Yoshida. 2020. EMTST: Engineering the Meta-theory of Session Types. In *TACAS (LNCS, Vol. 12079)*. 278–285.

Ornela Dardha and Simon J. Gay. 2018. A New Linear Logic for Deadlock-Free Session-Typed Processes. In *FOSSACS (LNCS, Vol. 10803)*. 91–109.

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

Derek Dreyer, Amal Ahmed, and Lars Birkedal. 2009. Logical Step-Indexed Logical Relations. In *LICS*. 71–80.

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/.

Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2018. ReLoC: A Mechanised Relational Logic for Fine-Grained Concurrency. In *LICS*. 442–451.

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), 895–930.

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

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

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

Paolo G. Giarrusso, Léo Stefanesco, Amin Timany, Lars Birkedal, and Robbert Krebbers. 2020. Scala step-by-step: soundness for DOT with step-indexed logical relations in Iris. *PACMPL* 4, ICFP (2020), 114:1–114:29.

Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. 2020a. Actris 2.0: Asynchronous session-type based reasoning in separation logic. (2020). https://itu.dk/people/jkas/papers/actris2.pdf Draft.

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

Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. 1998. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *ESOP (LNCS, Vol. 1381)*. 122–138.

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

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2020. 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*. 256–269.

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018b. Iris From the Ground Up: A Modular Foundation for Higher-Order Concurrent Separation Logic. *JFP* 28 (2018), e20.

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

Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. 2018. MoSeL: A General, Extensible Modal Framework for Interactive Proofs in Separation Logic. *PACMPL* 2, ICFP (2018), 77:1–77:30.

Robbert Krebbers, Ralf Jung, Ales Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017a. The Essence of Higher-Order Concurrent Separation Logic. In *ESOP (LNCS, Vol. 10201)*. 696–723.

Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017b. Interactive Proofs in Higher-Order Concurrent Separation Logic. In *POPL*. 205–217.

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

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

Dimitris Mostrous, Nobuko Yoshida, and Kohei Honda. 2009. Global Principal Typing in Partially Commutative Asynchronous Sessions. In *ESOP (LNCS, Vol. 5502)*. 316–332.

Jorge A. Pérez, Luís Caires, Frank Pfenning, and Bernardo Toninho. 2012. Linear Logical Relations for Session-Based Concurrency. In *ESOP (LNCS, Vol. 7211)*. 539–558.

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), 254–302.

Frank Pfenning and Conal Elliott. 1988. Higher-Order Abstract Syntax. In *PLDI*. 199–208.

Benjamin C. Pierce et al. 2020. Programming Language Foundations. https://softwarefoundations.cis.upenn.edu/plf-current/index.html

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

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

Joseph Tassarotti, Ralf Jung, and Robert Harper. 2017. A Higher-Order Logic for Concurrent Termination-Preserving Refinement. In *ESOP (LNCS, Vol. 10201)*. 909–936.

The Coq-std++ Team. 2020. An extended "standard library" for Coq. Available online at https://gitlab.mpi-sws.org/iris/stdpp.

Peter Thiemann. 2019. Intrinsically-Typed Mechanized Semantics for Session Types. In *PPDP*. 19:1–19:15.

Peter Thiemann and Vasco T. Vasconcelos. 2020. Label-dependent session types. *Proc. ACM Program. Lang.* 4, POPL (2020), 67:1–67:29.

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), 64:1–64:28.

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

Andrew K. Wright. 1995. Simple Imperative Polymorphism. *Lisp and Symbolic Computation* 8, 4 (1995), 343–355.

A 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 Coq, and can be found in [Anonymous Authors 2020].

In particular, the appendix shows the type and judgement definitions in Figure 7, the typing rules in Figures 8 and 9, and the subtyping rules in Figures 10 and 11.

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 (Appendix A.1). We furthermore describe kinded subtyping and type equivalence (Appendix A.2), shared reference types (Appendix A.3), and discuss the internal versions of all judgements of the type system (Appendix A.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). For example:

Ty-RefUniqLoad-Move

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

Ty-RefUniqLoad-Copy

$$\frac{\mathtt{copyable}\, A}{\Gamma, (x : \mathtt{ref}_{\mathtt{uniq}}\, A) \vDash {!x} : A \dashv \Gamma, (x : \mathtt{ref}_{\mathtt{uniq}}\, A)}$$

The full version of our type system unifies these rules as the single rule Ty-RefUniqLoad using the uncopy type former. This 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-Uncopy, this means that the uncopy type former has no effect on copyable types A, for which we have uncopy $A <: A$. However, when applied to a non-copyable type A, the uncopy type former cannot be stripped, preventing the value from being used again, similar 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 (□). 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.

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:

$$K <:> L \triangleq K <: L \wedge L <: K$$

Similar to the subtyping relation, the relation for type equivalence is kinded so it applies to both term and session types.

A.3 Shared References

We also have an additional type former $\mathtt{ref}_{\mathtt{shr}}$, 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 $\mathtt{ref}_{\mathtt{shr}}$ 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 $\mathtt{ref}_{\mathtt{uniq}}$.

A.4 Internal Judgements

In § 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} \longrightarrow \mathsf{iProp}\\
\mathsf{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)\\
\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) *\\
&\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\}\\
\mathsf{chan}\, S &\triangleq \lambda\, w.\, w \rightarrowtail S\\
\mathsf{copy}\, A &\triangleq \lambda\, w.\ \square(A\, w)\\
A \longrightarrow B &\triangleq \mathsf{copy}\ (A \multimap B)\\
\mathsf{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)) *\\
&\qquad\qquad \mathtt{isLock}\ lk\ (\exists v.\, (\ell \mapsto v) * \triangleright(A\, v))\\
\overline{\mathtt{mutex}}\, A &\triangleq \lambda\, w.\, \exists lk, \ell.\, (w = (lk, \ell)) * (\ell \mapsto -) *\\
&\qquad\qquad \mathtt{isLock}\ lk\ (\exists v.\, (\ell \mapsto v) * \triangleright(A\, v))
\end{aligned}
$$

Typing Judgement:

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

Session Types:

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

Subtyping:

$$
\begin{aligned}
A <: B &\triangleq \blacksquare(\forall v.\, A\ v \mathrel{-\!\!*} B\ v)\\
S <: T &\triangleq \blacksquare(S \sqsubseteq T)\\
K <:> L &\triangleq K <: L \wedge L <: K\\
\Gamma <:_{\mathbf{ctx}} \Gamma' &\triangleq \blacksquare(\forall \sigma.\, (\Gamma \vDash \sigma) \mathrel{-\!\!*} (\Gamma' \vDash \sigma))
\end{aligned}
$$

Other:

$$
\mathtt{copyable}\, A \triangleq A <: \mathsf{copy}\, A
$$

Figure 7. Typing judgements and type formers.

Basics:

$$\textsc{Ty-Unit}\quad \Gamma \vDash () : \mathbf{1} \dashv \Gamma$$

$$\textsc{Ty-Int}\quad \Gamma \vDash n : \mathbf{Z} \dashv \Gamma$$

$$\textsc{Ty-Bool}\quad \Gamma \vDash b : \mathbf{B} \dashv \Gamma$$

$$\textsc{Ty-Neg}\quad \frac{\Gamma \vDash e : \mathbf{B} \dashv \Gamma'}{\Gamma \vDash \neg e : \mathbf{B} \dashv \Gamma'}$$

$$\textsc{Ty-Arith}\quad \frac{\Gamma \vDash e_2 : \mathbf{Z} \dashv \Gamma' \qquad \Gamma' \vDash e_1 : \mathbf{Z} \dashv \Gamma'' \qquad \mathrm{op} \in \{+, -\}}{\Gamma \vDash e_1 \ \mathrm{op}\ e_2 : \mathbf{Z} \dashv \Gamma''}$$

$$\textsc{Ty-Cond}\quad \frac{\Gamma \vDash e_2 : \mathbf{Z} \dashv \Gamma' \qquad \Gamma' \vDash e_1 : \mathbf{Z} \dashv \Gamma'' \qquad \mathrm{op} \in \{=, \leq\}}{\Gamma \vDash e_1 \ \mathrm{op}\ e_2 : \mathbf{B} \dashv \Gamma''}$$

$$\textsc{Ty-If}\quad \frac{\Gamma \vDash e_1 : \mathbb{B} \dashv \Gamma' \qquad \Gamma' \vDash e_2 : A \dashv \Gamma'' \qquad \Gamma' \vDash e_3 : A \dashv \Gamma''}{\Gamma \vDash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : A \dashv \Gamma''}$$

$$\textsc{Ty-Var}\quad \Gamma, (x : A) \vDash x : A \dashv \Gamma, (x : \mathsf{uncopy}\ A)$$

$$\textsc{Ty-Lam}\quad \frac{\Gamma, (x : A) \vDash e : B \dashv \Gamma''}{\Gamma \cdot \Gamma' \vDash \lambda x.\, e : A \multimap B \dashv \Gamma'}$$

$$\textsc{Ty-Rec}\quad \frac{\begin{array}{c}\Gamma = (x_1 : A_1), \ldots, (x_n : A_n) \\ \Gamma_{\mathrm{copy}} = (x_1 : \mathsf{uncopy}\ A_1), \ldots, (x_n : \mathsf{uncopy}\ A_n) \\ \Gamma_{\mathrm{copy}}, (f : A \longrightarrow B), (x : A) \vDash e : B \dashv \Gamma''\end{array}}{\Gamma \cdot \Gamma' \vDash \texttt{rec}\ f\ x = e : A \longrightarrow B \dashv \Gamma'}$$

$$\textsc{Ty-App}\quad \frac{\Gamma \vDash e_2 : A \dashv \Gamma' \qquad \Gamma' \vDash e_1 : A \multimap B \dashv \Gamma''}{\Gamma \vDash e_1\ e_2 : B \dashv \Gamma''}$$

$$\textsc{Ty-Let}\quad \frac{\Gamma_1 \vDash e_1 : A \dashv \Gamma_2 \qquad \Gamma_2, (x : A) \vDash e_2 : B \dashv \Gamma_3}{\Gamma_1 \vDash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : B \dashv \Gamma_3 \setminus x}$$

$$\textsc{Ty-Par}\quad \frac{\Gamma_1 \vDash e_1 : A_1 \dashv \Gamma_1' \qquad \Gamma_2 \vDash e_2 : A_2 \dashv \Gamma_2'}{\Gamma_1 \cdot \Gamma_2 \vDash e_1 \mid\mid e_2 : A_1 \times A_2 \dashv \Gamma_1' \cdot \Gamma_2'}$$

$$\textsc{Ty-Sub}\quad \frac{\Gamma_1 <:_{\mathbf{ctx}} \Gamma_1' \qquad \Gamma_1' \vDash e : A \dashv \Gamma_2' \qquad A <: B \qquad \Gamma_2' <:_{\mathbf{ctx}} \Gamma_2}{\Gamma_1 \vDash e : B \dashv \Gamma_2}$$

Product and Sums:

$$\textsc{Ty-Pair}\quad \frac{\Gamma \vDash e_2 : A_2 \dashv \Gamma' \qquad \Gamma' \vDash e_1 : A_1 \dashv \Gamma''}{\Gamma \vDash (e_1, e_2) : A_1 \times A_2 \dashv \Gamma''}$$

$$\textsc{Ty-Fst}\quad \Gamma, (x : A_1 \times A_2) \vDash \texttt{fst}\ x : A_1 \dashv \Gamma, (x : \mathsf{uncopy}\ A_1 \times A_2)$$

$$\textsc{Ty-Snd}\quad \Gamma, (x : A_1 \times A_2) \vDash \texttt{snd}\ x : A_2 \dashv \Gamma, (x : A_1 \times \mathsf{uncopy}\ A_2)$$

$$\textsc{Ty-InL}\quad \frac{\Gamma \vDash e : A \dashv \Gamma'}{\Gamma \vDash \texttt{inl}\ e : A + B \dashv \Gamma'}$$

$$\textsc{Ty-InR}\quad \frac{\Gamma \vDash e : B \dashv \Gamma'}{\Gamma \vDash \texttt{inr}\ e : A + B \dashv \Gamma'}$$

$$\textsc{Ty-Case}\quad \frac{\Gamma \vDash e_1 : A + B \dashv \Gamma' \qquad \Gamma' \vDash e_2 : A \multimap C \dashv \Gamma'' \qquad \Gamma' \vDash e_3 : B \multimap C \dashv \Gamma''}{\Gamma \vDash \texttt{case}\ e_1\ e_2\ e_3 : C \dashv \Gamma''}$$

Polymorphism:

$$\textsc{Ty-TLam}\quad \frac{\Gamma \vDash e : A \dashv \Gamma'' \qquad X \notin FV(\Gamma, \Gamma')}{\Gamma \cdot \Gamma' \vDash \lambda\_.\, e : \forall X.\, A \dashv \Gamma'}$$

$$\textsc{Ty-TApp}\quad \frac{\Gamma \vDash e : \forall X.\, A \dashv \Gamma'}{\Gamma \vDash e\ () : A[K/X] \dashv \Gamma'}$$

$$\textsc{Ty-Pack}\quad \frac{\Gamma \vDash e : A[K/X] \dashv \Gamma'}{\Gamma \vDash e : \exists X.\, A \dashv \Gamma'}$$

$$\textsc{Ty-Unpack}\quad \frac{\Gamma \vDash e_1 : \exists X.\, A \dashv \Gamma' \qquad \Gamma', (x : A) \vDash e_2 : B \dashv \Gamma'' \qquad X \notin FV(\Gamma, B, \Gamma'')}{\Gamma \vDash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : B \dashv \Gamma'' \setminus x}$$

Figure 8. Term typing rules.

References:

Ty-RefAlloc

$$\frac{\Gamma \vDash e : A \dashv \Gamma'}{\Gamma \vDash \mathtt{ref}\ e : \mathsf{ref}_{\mathsf{uniq}}\, A \dashv \Gamma'}$$

Ty-RefFree

$$\frac{\Gamma \vDash e : \mathsf{ref}_{\mathsf{uniq}}\, A \dashv \Gamma'}{\Gamma \vDash \mathtt{free}\ e : \mathbf{1} \dashv \Gamma'}$$

Ty-RefUniqStore

$$\frac{\Gamma \vDash e : B \dashv \Gamma', (x : \mathsf{ref}_{\mathsf{uniq}}\, A)}{\Gamma \vDash x \ \leftarrow e : \mathbf{1} \dashv \Gamma', (x : \mathsf{ref}_{\mathsf{uniq}}\, B)}$$

Ty-RefUniqLoad

$$\Gamma, (x : \mathsf{ref}_{\mathsf{uniq}}\, A) \vDash\ !x : A \dashv \Gamma, (x : \mathsf{ref}_{\mathsf{uniq}}\, (\mathsf{uncopy}\, A))$$

Ty-ToRefShr

$$\frac{\Gamma \vDash e : \mathsf{ref}_{\mathsf{uniq}}\, (\mathsf{copy}\, A) \dashv \Gamma'}{\Gamma \vDash e : \mathsf{ref}_{\mathsf{shr}}\ A \dashv \Gamma'}$$

Ty-RefShr-Load

$$\frac{\Gamma \vDash e : \mathsf{ref}_{\mathsf{shr}}\ A \dashv \Gamma'}{\Gamma \vDash\ !e : A \dashv \Gamma'}$$

Ty-RefShrStore

$$\frac{\Gamma \vDash e_2 : \mathsf{copy}\, A \dashv \Gamma' \qquad \Gamma' \vDash e_1 : \mathsf{ref}_{\mathsf{shr}}\ A \dashv \Gamma''}{\Gamma \vDash e_1 \ \leftarrow e_2 : \mathbf{1} \dashv \Gamma''}$$

Locks:

Ty-MutexAlloc

$$\Gamma \vDash \mathsf{mutexalloc} : A \longrightarrow \mathsf{mutex}\, A \dashv \Gamma$$

Ty-MutexAcquire

$$\Gamma, (x : \mathsf{mutex}\, A) \vDash \mathsf{mutexacquire}\ x : A \dashv \Gamma, (x : \overline{\mathsf{mutex}}\, A)$$

Ty-MutexRelease

$$\frac{\Gamma \vDash e : A \dashv \Gamma', (x : \overline{\mathsf{mutex}}\, A)}{\Gamma \vDash \mathsf{mutexrelease}\ x\ e : \mathbf{1} \dashv \Gamma', (x : \mathsf{mutex}\, A)}$$

Channels:

Ty-ChanAlloc

$$\Gamma \vDash \mathtt{new\_chan} : \mathbf{1} \longrightarrow \mathsf{chan}\ S \times \mathsf{chan}\ \overline{S} \dashv \Gamma$$

Ty-ChanSend

$$\frac{\Gamma \vDash e : A \dashv \Gamma', (x : \mathsf{chan}\ (\mathbf{!}A.\, S))}{\Gamma \vDash \mathtt{send}\ x\ e : \mathbf{1} \dashv \Gamma', (x : \mathsf{chan}\ S)}$$

Ty-ChanRecv

$$\Gamma, (x : \mathsf{chan}\ (\mathbf{?}A.\, S)) \vDash \mathtt{recv}\ x : A \dashv \Gamma, (x : \mathsf{chan}\ S)$$

Ty-ChanRecvPoly

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

Ty-Select

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

Ty-Branch

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

Figure 9. 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
$$A <: \mathsf{any}$$

SubTy-Lolli
$$\frac{C <: A \qquad B <: D}{A \multimap B <: C \multimap D}$$

SubTy-Arr
$$\frac{C <: A \qquad B <: D}{A \rightarrow B <: C \rightarrow D}$$

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
$$\frac{A <: B}{\mathsf{ref}_{\mathsf{uniq}}\, A <: \mathsf{ref}_{\mathsf{uniq}}\, B}$$

SubTy-Ref-Shr
$$\frac{A <:> B}{\mathsf{ref}_{\mathsf{shr}}\, A <: \mathsf{ref}_{\mathsf{shr}}\, B}$$

SubTy-Mutex
$$\frac{A <:> B}{\mathsf{mutex}\, A <: \mathsf{mutex}\, B}$$

SubTy-MutexGuard
$$\frac{A <:> B}{\overline{\mathsf{mutex}}\, A <: \overline{\mathsf{mutex}}\, B}$$

SubTy-Chan
$$\frac{S <: T}{\mathsf{chan}\ S <: \mathsf{chan}\ T}$$

Context Subtyping:

Ctx-Permute
$$\frac{\Gamma' \text{ is a permutation of } \Gamma}{\Gamma <:_{\mathbf{ctx}} \Gamma'}$$

Ctx-Refl
$$\Gamma <:_{\mathbf{ctx}} \Gamma$$

Ctx-Trans
$$\frac{\Gamma_1 <:_{\mathbf{ctx}} \Gamma_2 \qquad \Gamma_2 <:_{\mathbf{ctx}} \Gamma_3}{\Gamma_1 <:_{\mathbf{ctx}} \Gamma_3}$$

Ctx-Nil
$$\Gamma <:_{\mathbf{ctx}} [\,]$$

Ctx-Cons
$$\frac{A <: B \qquad \Gamma <:_{\mathbf{ctx}} \Gamma'}{(x : A), \Gamma <:_{\mathbf{ctx}} (x : B), \Gamma'}$$

Ctx-App
$$\frac{\Gamma_1 <:_{\mathbf{ctx}} \Gamma_2 \qquad \Gamma_1' <:_{\mathbf{ctx}} \Gamma_2'}{\Gamma_1 \cdot \Gamma_1' <:_{\mathbf{ctx}} \Gamma_2 \cdot \Gamma_2'}$$

Ctx-Copy
$$(x : A) <:_{\mathbf{ctx}} (x : A), (x : \mathsf{uncopy}\, A)$$

Copyable Types:

SubTy-Copy
$$\frac{A <: B}{\mathsf{copy}\, A <: \mathsf{copy}\, B}$$

SubTy-Copy-Intro
$$\frac{\mathsf{copyable}\, A}{A <: \mathsf{copy}\, A}$$

SubTy-Copy-Elim
$$\mathsf{copy}\, A <: A$$

SubTy-Uncopy
$$\frac{A <: B}{\mathsf{uncopy}\, A <: \mathsf{uncopy}\, B}$$

SubTy-Uncopy-Intro
$$A <: \mathsf{uncopy}\, A$$

SubTy-Uncopy-Elim
$$\mathsf{uncopy}\ (\mathsf{copy}\, A) <: A$$

SubTy-Copyable-Copy
$$\mathsf{copyable}\ (\mathsf{copy}\, A)$$

SubTy-Copyable-Uncopy
$$\mathsf{copyable}\ (\mathsf{uncopy}\, A)$$

SubTy-Copyable-Any
$$\mathsf{copyable}\ \mathsf{any}$$

SubTy-Copyable-Unit
$$\mathsf{copyable}\ \mathbf{1}$$

SubTy-Copyable-Int
$$\mathsf{copyable}\ \mathbf{Z}$$

SubTy-Copyable-Bool
$$\mathsf{copyable}\ \mathbf{B}$$

SubTy-Copyable-Product
$$\frac{\mathsf{copyable}\, A \qquad \mathsf{copyable}\, B}{\mathsf{copyable}\ (A \times B)}$$

SubTy-Copyable-Sum
$$\frac{\mathsf{copyable}\, A \qquad \mathsf{copyable}\, B}{\mathsf{copyable}\ (A + B)}$$

SubTy-Copyable-Exists
$$\frac{\forall X.\ \mathsf{copyable}\, A}{\mathsf{copyable}\ (\exists X.\ A)}$$

SubTy-Copyable-RefShr
$$\mathsf{copyable}\ (\mathsf{ref}_{\mathsf{shr}}\ X)$$

Figure 10. Subtyping rules.

Session Subtyping:

SubTy-Send
$$\frac{B <: A \qquad S <: T}{!A.\,S <: \,!B.\,T}$$

SubTy-Recv
$$\frac{A <: B \qquad S <: T}{?A.\,S <: \,?B.\,T}$$

SubTy-Send-Intro
$$!_{(\vec{X}:\vec{k})}\,A.\,S <: \,!A[\vec{K}/\vec{X}].\,S[\vec{K}/\vec{X}]$$

SubTy-Recv-Intro
$$?A[\vec{K}/\vec{X}].\,S[\vec{K}/\vec{X}] <: \,?_{(\vec{X}:\vec{k})}\,A.\,S$$

SubTy-Send-Elim
$$\frac{S <: \,!A.\,T}{S <: \,!_{(\vec{X}:\vec{k})}\,A.\,T}$$

SubTy-Recv-Elim
$$\frac{?A.\,S <: T}{?_{(\vec{X}:\vec{k})}\,A.\,S <: T}$$

SubTy-Select
$$\frac{\forall i.\,\vec{S_i} <: \vec{T_i}}{\oplus\{\vec{l_i} : \vec{S_i}\}_{i\in\vec{i}} <: \oplus\{\vec{l_i} : \vec{T_i}\}_{i\in\vec{i}}}$$

SubTy-Select-SubsetEq
$$\frac{\vec{j} \subseteq \vec{i}}{\oplus\{\vec{l_i} : \vec{S_i}\}_{i\in\vec{i}} <: \oplus\{\vec{l_j} : \vec{S_j}\}_{j\in\vec{j}}}$$

SubTy-Branch
$$\frac{\forall i.\,\vec{S_i} <: \vec{T_i}}{\&\{\vec{l_i} : \vec{S_i}\}_{i\in\vec{i}} <: \&\{\vec{l_i} : \vec{T_i}\}_{i\in\vec{i}}}$$

SubTy-Branch-SubsetEq
$$\frac{\vec{i} \subseteq \vec{j}}{\&\{\vec{l_i} : \vec{S_i}\}_{i\in\vec{i}} <: \&\{\vec{l_j} : \vec{S_j}\}_{j\in\vec{j}}}$$

SubTy-Swap-Recv-Send
$$?A.\,!B.\,S <: \,!B.\,?A.\,S$$

SubTy-Swap-Branch-Send
$$\&\{l_1 : !A.\,S_1, \ldots, l_n : !A.\,S_n\} <: \,!A.\,\&\{l_1 : S_1, \ldots, l_n : S_n\}$$

SubTy-Swap-Recv-Select
$$?A.\,\oplus\{l_1 : S_1, \ldots, l_n : S_n\} <: \oplus\{l_1 : ?A.\,S_1, \ldots, l_n : ?A.\,S_n\}$$

SubTy-Swap-Branch-Select
$$\begin{array}{ll}
\&\{l_1 : \oplus\{l'_1 : S_{(1,1)}, \ldots, l'_m : S_{(1,m)}\}, & <: \oplus\{l'_1 : \&\{l_1 : S_{(1,1)}, \ldots, l_n : S_{(n,1)}\}, \\
\quad \ldots, & \quad \ldots, \\
\quad l_n : \oplus\{l'_1 : S_{(n,1)}, \ldots, l'_m : S_{(n,m)}\}\} & \quad l'_m : \&\{l_1 : S_{(n,1)}, \ldots, l_n : S_{(n,m)}\}\}
\end{array}$$

Append Subtyping:

SubTy-App
$$\frac{S <: U \qquad T <: V}{S \cdot T <: U \cdot V}$$

SubTy-App-Assoc
$$S \cdot (T \cdot U) <:> (S \cdot T) \cdot U$$

SubTy-App-Send
$$(!_{\vec{X}}\,A.\,S) \cdot T <:> \,!_{\vec{X}}\,A.\,(S \cdot T)$$

SubTy-App-Recv
$$(?_{\vec{X}}\,A.\,S) \cdot T <:> \,?_{\vec{X}}\,A.\,(S \cdot T)$$

SubTy-App-Select
$$(\oplus\{l_1 : S_1, \ldots, l_n : S_n\}) \cdot T <:> \oplus\{l_1 : S_1 \cdot T, \ldots, l_n : S_n \cdot T\}$$

SubTy-App-Branch
$$(\&\{l_1 : S_1, \ldots, l_n : S_n\}) \cdot T <:> \&\{l_1 : S_1 \cdot T, \ldots, l_n : S_n \cdot T\}$$

SubTy-App-End-Right
$$S \cdot \mathsf{end} <:> S$$

SubTy-App-End-Left
$$\mathsf{end} \cdot S <:> S$$

Duality Subtyping:

SubTy-Dual
$$\frac{T <: S}{\overline{\overline{S}} <: \overline{\overline{T}}}$$

SubTy-Dual-Left
$$\frac{\overline{T} <: S}{\overline{\overline{S}} <: T}$$

SubTy-Dual-Right
$$\frac{T <: \overline{S}}{S <: \overline{\overline{T}}}$$

SubTy-Dual-Send
$$\overline{!_{\vec{X}}\,A.\,S} <:> \,?_{\vec{X}}\,A.\,\overline{S}$$

SubTy-Dual-Recv
$$\overline{?_{\vec{X}}\,A.\,S} <:> \,!_{\vec{X}}\,A.\,\overline{S}$$

SubTy-Dual-Select
$$\overline{\oplus\{l_1 : S_1, \ldots, l_n : S_n\}} <:> \&\{l_1 : \overline{S_1}, \ldots, l_n : \overline{S_n}\}$$

SubTy-Dual-Branch
$$\overline{\&\{l_1 : S_1, \ldots, l_n : S_n\}} <:> \oplus\{l_1 : \overline{S_1}, \ldots, l_n : \overline{S_n}\}$$

SubTy-Dual-End
$$\overline{\mathsf{end}} <:> \mathsf{end}$$

Figure 11. Subtyping rules (cont.)