Mechanised Semantic Session Typing

Jonas Kastberg Hinrichsen, IT University of Copenhagen

Joint work with
Daniël Louwrink, University of Amsterdam
Robbert Krebbers, Delft University of Technology
Jesper Bengtson, IT University of Copenhagen

04 June 2020
VEST
Problem

Mechanising session types is hard
Problem

Mechanising session types is hard

- **Linearity** requires explicit handling
Problem

Mechanising session types is hard

- **Linearity** requires explicit handling
- **Binders** impose non-trivial proof effort
Problem

Mechanising session types is hard

- **Linearity** requires explicit handling
- **Binders** impose non-trivial proof effort
- **Extensions** impose immodular proof effort
Mechanising session types is hard, especially for **syntactic type systems**

- **Linearity** requires explicit handling
- **Binders** impose non-trivial proof effort
- **Extensions** impose immodular proof effort
Shortcomings of Syntactic Typing

In a *syntactic type system*
In a **syntactic type system**

- **Types** defined as a closed inductive definition
Shortcomings of Syntactic Typing

In a **syntactic type system**

- **Types** defined as a closed inductive definition
- **Rules** defined as a closed inductive relation
Shortcomings of Syntactic Typing

In a syntactic type system:

- **Types** defined as a closed inductive definition
- **Rules** defined as a closed inductive relation
- **Soundness** proven as **progress** and **preservation**
Shortcomings of Syntactic Typing

In a syntactic type system

- **Types** defined as a closed inductive definition
- **Rules** defined as a closed inductive relation
- **Soundness** proven as **progress** and **preservation** using induction over the relation
Shortcomings of Syntactic Typing

In a **syntactic type system**

- **Types** defined as a closed inductive definition
- **Rules** defined as a closed inductive relation
- **Soundness** proven as *progress* and *preservation* using induction over the relation

**Linearity** requires explicit handling

- Explicit context splitting in rules
Shortcomings of Syntactic Typing

In a **syntactic type system**

- **Types** defined as a closed inductive definition
- **Rules** defined as a closed inductive relation
- **Soundness** proven as **progress** and **preservation** using induction over the relation

**Linearity** requires explicit handling

- Explicit context splitting in rules

**Binders** impose non-trivial proof effort

- Manual capture-avoiding substitution/renaming
Shortcomings of Syntactic Typing

In a *syntactic type system*
- **Types** defined as a closed inductive definition
- **Rules** defined as a closed inductive relation
- **Soundness** proven as *progress* and *preservation* using induction over the relation

**Linearity** requires explicit handling
- Explicit context splitting in rules

**Binders** impose non-trivial proof effort
- Manual capture-avoiding substitution/renaming

**Extensions** impose immodular proof effort
- Must reprove *progress* and *preservation* when adding types/rules
Goal:
A “mechanisable” session type system
Solution:
A semantic session type system!
Solution - Semantic Typing!

A **semantic type system** is defined in terms of the language semantics:

\[ Z \triangleq \lambda w. w \in Z \]

**Judgement** defined as safety-capturing evaluation: \( \Gamma \models e : A \) if \( e \) does not get stuck and if \( e \) reduces to a value \( v \), \( A v \) holds.

**Rules** are proven as lemmas: \( \models i : Z \Rightarrow i \in Z \)

**Soundness** is a consequence of the judgement definition

Linearity and binders can be inherited from underlying logic

Extensions can be added modularly

\[ B \triangleq \lambda w. w \in B \]
A **semantic type system** is defined in terms of the language semantics:

- **Types** defined as predicates over values: $\mathbb{Z} \triangleq \lambda w. w \in \mathbb{Z}$
A **semantic type system** is defined in terms of the language semantics:

- **Types** defined as predicates over values: \( \mathbb{Z} \triangleq \lambda w. w \in \mathbb{Z} \)
- **Judgement** defined as safety-capturing evaluation: \( \Gamma \vdash e : A \)
A semantic type system is defined in terms of the language semantics:

- **Types** defined as predicates over values: \( \mathcal{Z} \triangleq \lambda w. w \in \mathbb{Z} \)
- **Judgement** defined as safety-capturing evaluation: \( \Gamma \vdash e : A \)
  
  \( e \) does not get *stuck*
A **semantic type system** is defined in terms of the language semantics:

- **Types** defined as predicates over values: \( \mathbb{Z} \triangleq \lambda w. w \in \mathbb{Z} \)
- **Judgement** defined as safety-capturing evaluation: \( \Gamma \vdash e : A \)
  
  \( e \) does not get *stuck* and if \( e \) reduces to a value \( v \), \( A v \) holds.
A semantic type system is defined in terms of the language semantics:

- **Types** defined as predicates over values: \( Z \triangleq \lambda w. w \in Z \)
- **Judgement** defined as safety-capturing evaluation: \( \Gamma \vdash e : A \)  
  
  - \( e \) does not get stuck and if \( e \) reduces to a value \( v \), \( A v \) holds.
- **Rules** are proven as lemmas
A semantic type system is defined in terms of the language semantics:

- **Types** defined as predicates over values: \( Z \triangleq \lambda w. \ w \in Z \)
- **Judgement** defined as safety-capturing evaluation: \( \Gamma \vdash e : A \) 
  
  - \( e \) does not get stuck and if \( e \) reduces to a value \( v \), \( A v \) holds.
- **Rules** are proven as lemmas: \( \vdash i : Z \)
A **semantic type system** is defined in terms of the language semantics:

- **Types** defined as predicates over values: \( Z \triangleq \lambda w. w \in \mathbb{Z} \)
- **Judgement** defined as safety-capturing evaluation: \( \Gamma \vdash e : A \)
  - \( e \) does not get *stuck* and if \( e \) reduces to a value \( v \), \( A v \) holds.
- **Rules** are proven as lemmas: \( \vdash i : Z \leadsto i \in \mathbb{Z} \)
A **semantic type system** is defined in terms of the language semantics:

- **Types** defined as predicates over values: \( \mathbb{Z} \triangleq \lambda w. w \in \mathbb{Z} \)
- **Judgement** defined as safety-capturing evaluation: \( \Gamma \vdash e : A \)
  - \( e \) does not get stuck and if \( e \) reduces to a value \( v \), \( A v \) holds.
- **Rules** are proven as lemmas: \( \vdash i : \mathbb{Z} \leadsto i \in \mathbb{Z} \)
- **Soundness** is a consequence of the judgement definition
A **semantic type system** is defined in terms of the language semantics:

- **Types** defined as predicates over values: $\mathcal{Z} \triangleq \lambda w. w \in \mathbb{Z}$

- **Judgement** defined as safety-capturing evaluation: $\Gamma \vdash e : A$
  
  $e$ does not get stuck and if $e$ reduces to a value $v$, $A v$ holds.

- **Rules** are proven as lemmas: $\vdash i : \mathcal{Z} \rightsquigarrow i \in \mathbb{Z}$

- **Soundness** is a consequence of the judgement definition

**Linearity** and **binders** can be inherited from underlying logic
A **semantic type system** is defined in terms of the language semantics:

- **Types** defined as predicates over values: \( \mathbb{Z} \triangleq \lambda w. w \in \mathbb{Z} \)
- **Judgement** defined as safety-capturing evaluation: \( \Gamma \vdash e : A \)
  
  - \( e \) does not get stuck
  - and
  - if \( e \) reduces to a value \( v \), \( A v \) holds.
- **Rules** are proven as lemmas: \( \vdash i : \mathbb{Z} \rightsquigarrow \text{true} \Rightarrow i \in \mathbb{Z} \)
- **Soundness** is a consequence of the judgement definition

**Linearity** and **binders** can be inherited from underlying logic

**Extensions** can be added modularly
A semantic type system is defined in terms of the language semantics:

- **Types** defined as predicates over values: \( \mathbb{Z} \triangleq \lambda w. w \in \mathbb{Z} \)
- **Judgement** defined as safety-capturing evaluation: \( \Gamma \vdash e : A \)
  
  e does not get stuck and if e reduces to a value v, \( A v \) holds.
- **Rules** are proven as lemmas: \( \vdash i : \mathbb{Z} \rightsquigarrow i \in \mathbb{Z} \)
- **Soundness** is a consequence of the judgement definition

**Linearity** and binders can be inherited from underlying logic

**Extensions** can be added modularly

- Adding types and rules does not inherently impose new proof effort on existing types, rules and soundness
Solution - Semantic Typing!

A semantic type system is defined in terms of the language semantics:

- **Types** defined as predicates over values: \( \mathbb{Z} \triangleq \lambda w. w \in \mathbb{Z} \)
- **Judgement** defined as safety-capturing evaluation: \( \Gamma \vdash e : A \)
  - \( e \) does not get stuck and if \( e \) reduces to a value \( v \), \( A v \) holds.
- **Rules** are proven as lemmas: \( \vdash i : \mathbb{Z} \leadsto i \in \mathbb{Z} \)
- **Soundness** is a consequence of the judgement definition

**Linearity** and **binders** can be inherited from underlying logic

**Extensions** can be added modularly

- Adding types and rules does not inherently impose new proof effort on existing types, rules and soundness

\[
\mathbb{B} \triangleq \lambda w. w \in \mathbb{B}
\]
Solution - Semantic Typing!

A **semantic type system** is defined in terms of the language semantics:

- **Types** defined as predicates over values: \( \mathbb{Z} \triangleq \lambda w. w \in \mathbb{Z} \)
- **Judgement** defined as safety-capturing evaluation: \( \Gamma \vdash e : A \) if \( e \) does not get stuck and if \( e \) reduces to a value \( v \), \( A v \) holds.
- **Rules** are proven as lemmas: \( \vdash i : \mathbb{Z} \leadsto i \in \mathbb{Z} \)
- **Soundness** is a consequence of the judgement definition

**Linearity** and **binders** can be inherited from underlying logic

**Extensions** can be added modularly

- Adding types and rules does not inherently impose new proof effort on existing types, rules and soundness

\[
\mathbb{B} \triangleq \lambda w. w \in \mathbb{B} \quad \vdash b : \mathbb{B}
\]
Key Idea

Semantic Typing

Semantic Typing [Milner, Princeton Proof-Carrying Code project, RustBelt Project]

- **Linearity** and **binders** can be inherited from underlying logic
- **Extensions** can be added modularly
Key Idea

**Semantic Typing using Iris**

**Semantic Typing** [Milner, Princeton Proof-Carrying Code project, RustBelt Project]

- **Linearity** and **binders** can be inherited from underlying logic
- **Extensions** can be added modularly

**Iris** [Iris project]

- **Higher-Order**: Recursion, Polymorphism
- **Concurrent**: Ghost state mechanisms to reason about concurrency
- **Separation Logic**: Implicit separation of **linear** ownership
- Mechanised in **Coq** (which has **binder** support)
Key Idea

**Semantic Typing** using **Iris** and **Actris**

**Semantic Typing** [Milner, Princeton Proof-Carrying Code project, RustBelt Project]

- **Linearity** and **binders** can be inherited from underlying logic
- **Extensions** can be added modularly

**Iris** [Iris project]

- **Higher-Order**: Recursion, Polymorphism
- **Concurrent**: Ghost state mechanisms to reason about concurrency
- **Separation Logic**: Implicit separation of **linear** ownership
- Mechanised in **Coq** (which has **binder** support)

**Actris** [Hinrichsen et al., POPL’20]

- **Dependent separation protocols (DSP)**: Session type-style logical protocols
- Mechanised in **Coq**
Contributions

Semantic Session Type System

- Rich extensible type system for session types
  - Term and session type equi-recursion
  - Term and session type polymorphism
  - Term and (asynchronous) session type subtyping
  - Unique and shared reference types, Copyable types, Lock types
- Full mechanisation in Coq (https://gitlab.mpi-sws.org/iris/actris)
- Supports integrating safe yet untypeable programs
- Actris 2.0: Subprotocols
Semantic Session Type System
Language

**Language**: ML-like language extended with concurrency, state and message passing

\[ e \in \text{Expr} ::= v \mid x \mid \text{rec } f(x) = e \mid e_1(e_2) \mid e_1 \mid e_2 \mid \text{ref } (e) \mid !e \mid e_1 \leftarrow e_2 \mid \text{new} \_\text{chan} () \mid \text{send } e_1 \ e_2 \mid \text{recv } e \mid \ldots \]
Language

**Language:** ML-like language extended with concurrency, state and message passing

\[ e \in \text{Expr} ::= v \mid x \mid \text{rec } f(x) = e \mid e_1(e_2) \mid e_1 \parallel e_2 \mid \text{ref } (e) \mid ! e \mid e_1 \leftarrow e_2 \mid \text{new\_chan } () \mid \text{send } e_1 e_2 \mid \text{recv } e \mid \ldots \]

Only allows substitution with closed terms

- To avoid substitution overhead
Language: ML-like language extended with concurrency, state and message passing

\[ e \in \text{Expr} ::= v \mid x \mid \text{rec } f(x) = e \mid e_1(e_2) \mid e_1 \parallel e_2 \mid \text{ref } (e) \mid !e \mid e_1 \leftarrow e_2 \]
\[ \text{new chan } () \mid \text{send } e_1 \ e_2 \mid \text{recv } e \mid \ldots \]

Only allows substitution with closed terms

- To avoid substitution overhead

Evaluation is performed right-to-left

- To allow side-effects in function applications (e.g. \text{send } c (\text{recv } c))
Language: ML-like language extended with concurrency, state and message passing

\[ e \in \text{Expr} ::= v \mid x \mid \text{rec } f(x) = e \mid e_1(e_2) \mid e_1 || e_2 \mid \text{ref } (e) \mid ! e \mid e_1 \leftarrow e_2 \mid \text{new\_chan } () \mid \text{send } e_1 e_2 \mid \text{recv } e \mid \ldots \]

Only allows substitution with closed terms

- To avoid substitution overhead

Evaluation is performed right-to-left

- To allow side-effects in function applications (e.g. \textit{send} \(c\) (\textit{recv} \(c\))

Message-passing is:

- **Binary**: Each channel have one pair of endpoints
- **Asynchronous**: \textit{send} does not block, two buffers per endpoint pair
- **Affine**: No \textit{close} expression, channels can be thrown away
Semantic Term Types

**Types** as Iris predicates:

\[
\text{Type}^\star \triangleq \text{Val} \rightarrow \text{iProp}
\]
Semantic Term Types

**Types** as Iris predicates:

\[
\begin{align*}
\text{Type} & \triangleq \text{Val} \rightarrow \text{iProp} \\
\mathcal{Z} & \triangleq \lambda w. w \in \mathbb{Z}
\end{align*}
\]
Semantic Term Types

**Types** as Iris predicates:

- \( \text{Type}^\star \triangleq \text{Val} \rightarrow \text{iProp} \)
- \( \mathbb{Z} \triangleq \lambda w. \ w \in \mathbb{Z} \)
- \( A_1 \times A_2 \triangleq \lambda w. \ \exists w_1, w_2. \ w = (w_1, w_2) \triangleright (A_1 \ w_1) \triangleright (A_2 \ w_2) \)
Types as Iris predicates:

\[
\begin{align*}
\text{Type}^* & \triangleq \text{Val} \rightarrow \text{iProp} \\
\mathbb{Z} & \triangleq \lambda w. w \in \mathbb{Z} \\
A_1 \times A_2 & \triangleq \lambda w. \exists w_1, w_2. w = (w_1, w_2) \triangleright (A_1 w_1) \triangleright (A_2 w_2) \\
A \mapsto B & \triangleq \lambda w. \forall v. \triangleright (A v) \triangleright \text{wp} (w v) \{ B \}
\end{align*}
\]
Semantic Term Types

**Types** as Iris predicates:

\[
\begin{align*}
\text{Type}\star & \triangleq \text{Val} \rightarrow \text{iProp} \\
\mathbb{Z} & \triangleq \lambda w. w \in \mathbb{Z} \\
A_1 \times A_2 & \triangleq \lambda w. \exists w_1, w_2. w = (w_1, w_2) \triangleright (A_1 w_1) \triangleright (A_2 w_2) \\
A \circ B & \triangleq \lambda w. \forall v. \triangleright (A v) \rightsquigarrow \wp (w, v) \{B\}
\end{align*}
\]

**Judgement** as Iris weakest precondition:

\[
\Gamma \triangleright e : A \triangleq \Gamma' \triangleq \forall \sigma. (\Gamma \triangleright \sigma) \rightsquigarrow \wp e[\sigma] \{v. \, A \, v \ast (\Gamma' \triangleright \sigma)\}
\]

---

\(\wp e \{v. \Phi\}\) dictates \(e\) does not get stuck and if \(e\) reduces to a value \(v\) then \(\Phi \, v\) holds
Semantic Term Types

**Types** as Iris predicates:

\[\text{Type} \star \triangleq \text{Val} \rightarrow \text{iProp} \]
\[Z \triangleq \lambda w. w \in \mathbb{Z}\]
\[A_1 \times A_2 \triangleq \lambda w. \exists w_1, w_2. w = (w_1, w_2) \triangleright (A_1 w_1) \triangleright (A_2 w_2)\]
\[A \rightarrow B \triangleq \lambda w. \forall v. \triangleright (A v) \rightarrow \wp (w v) \{B\}\]

**Judgement** as Iris weakest precondition:

\[\Gamma \models e : A \models \Gamma' \triangleq \forall \sigma. (\Gamma \models \sigma) \rightarrow \wp e[\sigma] \{v. A v \star (\Gamma' \models \sigma)\}\]

**Soundness:** If \(\emptyset \models e : A \models \Gamma\) then \(e\) does not get stuck

- Consequence of Iris’s adequacy of weakest precondition

---

\[\wp e \{v.\Phi\} \text{ dictates } e \text{ does not get stuck and if } e \text{ reduces to a value } v \text{ then } \Phi v \text{ holds}\]
Semantic Term Types - Proofs

**Rules:**

\[ \Gamma \models i : Z \]

\[ \Gamma_2 \models e_1 : A_1 \models \Gamma_3 \quad \Gamma_1 \models e_2 : A_2 \models \Gamma_2 \]

\[ \Gamma_1 \models (e_1, e_2) : A_1 \times A_2 \models \Gamma_3 \]

If \( \emptyset \models e : A \models \Gamma \) then \( e \) does not get stuck

**Proofs:**

Lemma 1: \( \text{ltyped\_int} \) \( \Gamma (i : Z) : \vdash \Gamma \models \mathbf{#}i : \text{lt\_int} \).

Proof: iIntros "!>" (vs) "Henv /=". iApply wp\_value. eauto. Qed.

Lemma 2: \( \text{ltyped\_pair} \) \( \Gamma_1 \Gamma_2 \Gamma_3 e_1 e_2 A_1 A_2 : \)

\[ (\Gamma_2 = e_1 : A_1 \models \Gamma_3) \rightarrow (\Gamma_1 = e_2 : A_2 \models \Gamma_2) \rightarrow \]

\[ \Gamma_1 = (e_1,e_2) : A_1 * A_2 \models \Gamma_3. \]

Proof:

iIntros "#H1 #H2". iIntros (vs) "!> HGamma /=".

wp\_apply (wp\_wand with "(H2 [HGamma //])"); iIntros (w2) "[HA2 HGamma]".

wp\_apply (wp\_wand with "(H1 [HGamma //])"); iIntros (w1) "[HA1 HGamma]".

wp\_pures. iFrame "HGamma". iExists w1, w2. by iFrame.

Qed.

Lemma 3: \( \text{ltyped\_safety} \) \( \{\text{heapPreG} \Sigma\} e\;a\;es\;e'\;e'' : \)

\[ (\forall \{\text{heapG} \Sigma\}, \exists A \Gamma', \vdash o = e : A \models \Gamma') \rightarrow \]

rtc\_erased\_step ([e], o) (es, e') = e' \in es \rightarrow

is\_Some (to\_val e') \lor \text{reducible } e' \forall o'.

Proof:

intros Hty. apply (heap\_adequacy \Sigma NotStuck e o (\& _, True))=> // ?.

destruct (Hty _) as (A & \Gamma & He). iIntros "_".

destruct (He $!\varnothing with "[]") as "He"; first by rewrite /env\_ltyped.

eval (rewrite -(subst\_map\_empty e)). iApply (wp\_wand with "He"); auto.

Qed.
But what about session types?
Session types as a new type kind:

\[
\text{Type}^\diamondsuit \triangleq \, ? \\
!A. \, S \triangleq \, ? \\
?A. \, S \triangleq \, ? \\
\text{end} \triangleq \, ?
\]

\[
\text{Type}^\bigstar \triangleq \text{Val} \rightarrow \text{iProp} \\
\text{chan} \, S \triangleq \lambda w. \, ?
\]

Requires capturing:

- **Linearity** of channel endpoint ownership
- **Delegation** of linear types / channels
- **Session fidelity** of communicated messages
### Session type-inspired protocols for functional correctness

<table>
<thead>
<tr>
<th>Example</th>
<th>Dependent separation protocols</th>
<th>Syntactic session types</th>
</tr>
</thead>
<tbody>
<tr>
<td>?(x : \mathbb{Z}) \langle x \rangle { x &gt; 10 }. ?\langle x + 10 \rangle { True }. end</td>
<td>?\mathbb{Z}. ?\mathbb{Z}. end</td>
<td></td>
</tr>
<tr>
<td>c \mapsto prot</td>
<td>c : S</td>
<td></td>
</tr>
</tbody>
</table>

**Usage**

*c* $\mapsto$ *prot*
Session types as dependent separation protocols:

\[
\begin{align*}
\text{Type} & \triangleq \text{iProto} \\
!A. S & \triangleq !(v : \text{Val}) \langle v \rangle \{\triangleright (A \, v)\}. S \\
?A. S & \triangleq ?(v : \text{Val}) \langle v \rangle \{\triangleright (A \, v)\}. S \\
\text{end} & \triangleq \text{end}
\end{align*}
\]

Dependent separation protocols:

- **Example:**  
  \( ?(x : \mathbb{Z}) \langle x \rangle \{x > 10\}. ?\langle x + 10\rangle\{\text{True}\}. \text{end} \)

- **Usage:**  
  \( c \mapsto \text{prot} \)
Semantic Session Types - Rules

Rules are proven as lemmas using the rules for dependent separation protocols

\[
\Gamma \vdash \text{new}_\text{chan}() : \text{chan } S \times \text{chan } S' \vdash \Gamma
\]

\[
\Gamma, (c : \text{chan !}A. S), (x : A) \vdash \text{send } c x : 1 \quad \vdash \Gamma, (c : \text{chan } S)
\]

\[
\Gamma, (c : \text{chan (?)}A. S) \vdash \text{recv } c : A \quad \vdash \Gamma, (c : \text{chan } S)
\]
Semantic Session Types - Proofs

Rule:

\[ \Gamma, (c: \text{chan } (?A. S)) \vdash \text{recv } c : A \equiv \Gamma, (c: \text{chan } S) \]

Proof:

**Lemma** \text{ltyped_recv} \( \Gamma (x : \text{string}) A S : \)
\( \Gamma \xrightarrow{\text{ ! }} x = \text{Some (chan (<??> TY A; S))\%lty} \rightarrow \)
\( \vdash \Gamma \vdash \text{recv } x : A = \langle x:=(\text{chan } S)\%lty] > \Gamma. \)

**Proof.**

iIntros (Hx) "!>". iIntros (vs) "H\Gamma"=> /=.
iDestruct (env\_ltyped\_lookup _ _ _ _ (Hx) with "H\Gamma") as (v' Heq) "[Hc H\Gamma]".
\text{rewrite Heq.}
wp\_recv (v) as "HA". iFrame "HA".
iDestruct (env\_ltyped\_insert _ _ x (chan _) _ with "[Hc //] H\Gamma") as "H\Gamma"=> /=.
\text{by rewrite insert\_delete (insert\_id vs).}

Qed.
Extensions
Overview of features

Iris and Actris gives immediate rise to many type features

Subprotocols:
prot₁ ⊑ prot₂
▶ Generalisation of asynchronous subtyping for functional correctness
▶ Makes asynchronous semantics explicit by swap rule

prot ⊑ !⟨v₂⟩{P₂}.

prot ⊑ ?⟨v₁⟩{P₁}.

S <:
!A₂.

?A₁.

Non-trivial extension due to dependent binders and step-indexing
▶ Required updates to the model of iProto
Overview of features

Iris and Actris gives immediate rise to many type features

| Linear products | Separation Conjunction (⋆) |

Subprotocols:

\[
\text{prot}_1 \sqsubseteq \text{prot}_2
\]

▶ Generalisation of asynchronous subtyping for functional correctness
▶ Makes asynchronous semantics explicit by swap rule

\[
\langle v_1 \rangle \{ P_1 \} . ! \langle v_2 \rangle \{ P_2 \} . \text{prot} \sqsubseteq ! \langle v_2 \rangle \{ P_2 \} . ? \langle v_1 \rangle \{ P_1 \} . \text{prot}
\]

▶ Non-trivial extension due to dependent binders and step-indexing
▶ Required updates to the model of iProto
Overview of features

Iris and Actris gives immediate rise to many type features

<table>
<thead>
<tr>
<th>Linear products</th>
<th>Separation Conjunction ((\ast))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function types</td>
<td>Wand ((\neg\ast)) and weakest precondition (wp e {\Phi})</td>
</tr>
</tbody>
</table>
Overview of features

Iris and Actris gives immediate rise to many type features

<table>
<thead>
<tr>
<th>Linear products</th>
<th>Separation Conjunction (*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function types</td>
<td>Wand (−*) and weakest precondition (wp e {Φ})</td>
</tr>
<tr>
<td>Session types</td>
<td>Actris dependent separation protocols (iProto)</td>
</tr>
</tbody>
</table>
Iris and Actris gives immediate rise to many type features

<table>
<thead>
<tr>
<th>Feature Type</th>
<th>Feature Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear products</td>
<td>Separation Conjunction ((*))</td>
</tr>
<tr>
<td>Function types</td>
<td>Wand ((\rightarrow)) and weakest precondition ((wp e {\Phi}))</td>
</tr>
<tr>
<td>Session types</td>
<td>Actris dependent separation protocols (iProto)</td>
</tr>
<tr>
<td>Unique references</td>
<td>Points-to connective ((\ell \mapsto v))</td>
</tr>
</tbody>
</table>
Overview of features

Iris and Actris gives immediate rise to many type features

<table>
<thead>
<tr>
<th>Feature Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear products</td>
<td>Separation Conjunction (★)</td>
</tr>
<tr>
<td>Function types</td>
<td>Wand (→*) and weakest precondition (wp e {Φ})</td>
</tr>
<tr>
<td>Session types</td>
<td>Actris dependent separation protocols (iProto)</td>
</tr>
<tr>
<td>Unique references</td>
<td>Points-to connective (ℓ ↦→ v)</td>
</tr>
<tr>
<td>Shared references</td>
<td>Invariants ([P])</td>
</tr>
</tbody>
</table>

Subprotocols:

\[\text{prot}_1 \triangleright \text{prot}_2\]

Generalisation of asynchronous subtyping for functional correctness

- Makes asynchronous semantics explicit by swap rule
- \[\langle v_1 \rangle \{P_1\}.!\langle v_2 \rangle \{P_2\}.	ext{prot} \triangleright !\langle v_2 \rangle \{P_2\}..\langle v_1 \rangle \{P_1\}.	ext{prot}\]

- Non-trivial extension due to dependent binders and step-indexing
- Required updates to the model of iProto
Overview of features

Iris and Actris gives immediate rise to many type features

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear products</td>
<td>Separation Conjunction (*)</td>
</tr>
<tr>
<td>Function types</td>
<td>Wand ((\rightsquigarrow)) and weakest precondition ((\text{wp} e {\Phi}))</td>
</tr>
<tr>
<td>Session types</td>
<td>Actris dependent separation protocols (iProto)</td>
</tr>
<tr>
<td>Unique references</td>
<td>Points-to connective ((\ell \mapsto v))</td>
</tr>
<tr>
<td>Shared references</td>
<td>Invariants (([P]))</td>
</tr>
<tr>
<td>Copyable types</td>
<td>Persistent modality ((\square))</td>
</tr>
</tbody>
</table>
Overview of features

**Iris** and **Actris** gives immediate rise to many type features

<table>
<thead>
<tr>
<th>Feature Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear products</td>
<td>Separation Conjunction (*)&amp;</td>
</tr>
<tr>
<td>Function types</td>
<td>Wand (→*) and weakest precondition (wp e {Φ})</td>
</tr>
<tr>
<td>Session types</td>
<td>Actris dependent separation protocols (iProto)</td>
</tr>
<tr>
<td>Unique references</td>
<td>Points-to connective (ℓ ↦→ v)</td>
</tr>
<tr>
<td>Shared references</td>
<td>Invariants ([P])</td>
</tr>
<tr>
<td>Copyable types</td>
<td>Persistent modality (□)</td>
</tr>
<tr>
<td>Lock types</td>
<td>Iris’s lock library</td>
</tr>
</tbody>
</table>
Overview of features

Iris and Actris gives immediate rise to many type features

<table>
<thead>
<tr>
<th>Linear products</th>
<th>Separation Conjunction ((\star))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function types</td>
<td>Wand ((&lt;\star)) and weakest precondition (wp e ({\Phi}))</td>
</tr>
<tr>
<td>Session types</td>
<td>Actris dependent separation protocols (iProto)</td>
</tr>
<tr>
<td>Unique references</td>
<td>Points-to connective ((\ell \mapsto v))</td>
</tr>
<tr>
<td>Shared references</td>
<td>Invariants (([P]))</td>
</tr>
<tr>
<td>Copyable types</td>
<td>Persistent modality ((\square))</td>
</tr>
<tr>
<td>Lock types</td>
<td>Iris’s lock library</td>
</tr>
<tr>
<td>Session choice types</td>
<td>Actris dependent separation protocols (iProto)</td>
</tr>
</tbody>
</table>
Overview of features

*Iris* and *Actris* gives immediate rise to many type features

<table>
<thead>
<tr>
<th>Linear products</th>
<th>Separation Conjunction ((\ast))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function types</td>
<td>Wand ((-\ast)) and weakest precondition ((\text{wp e } {\Phi}))</td>
</tr>
<tr>
<td>Session types</td>
<td>Actris dependent separation protocols (iProto)</td>
</tr>
<tr>
<td>Unique references</td>
<td>Points-to connective ((\ell \mapsto v))</td>
</tr>
<tr>
<td>Shared references</td>
<td>Invariants ((P))</td>
</tr>
<tr>
<td>Copyable types</td>
<td>Persistent modality ((\Box))</td>
</tr>
<tr>
<td>Lock types</td>
<td>Iris’s lock library</td>
</tr>
<tr>
<td>Session choice types</td>
<td>Actris dependent separation protocols (iProto)</td>
</tr>
<tr>
<td>Recursion</td>
<td>Guarded step-indexed recursion ((\triangleright))</td>
</tr>
</tbody>
</table>
Overview of features

**Iris** and **Actris** gives immediate rise to many type features

<table>
<thead>
<tr>
<th>Feature Type</th>
<th>Iris/Actris Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear products</td>
<td>Separation Conjunction (*)</td>
</tr>
<tr>
<td>Function types</td>
<td>Wand (−∗) and weakest precondition (wp e {Φ})</td>
</tr>
<tr>
<td>Session types</td>
<td>Actris dependent separation protocols (iProto)</td>
</tr>
<tr>
<td>Unique references</td>
<td>Points-to connective (ℓ ↦→ v)</td>
</tr>
<tr>
<td>Shared references</td>
<td>Invariants ([P])</td>
</tr>
<tr>
<td>Copyable types</td>
<td>Persistent modality (□)</td>
</tr>
<tr>
<td>Lock types</td>
<td>Iris’s lock library</td>
</tr>
<tr>
<td>Session choice types</td>
<td>Actris dependent separation protocols (iProto)</td>
</tr>
<tr>
<td>Recursion</td>
<td>Guarded step-indexed recursion (▲)</td>
</tr>
<tr>
<td>Term polymorphism</td>
<td>Higher-order impredicative quantifiers</td>
</tr>
</tbody>
</table>
Overview of features

**Iris** and **Actris** gives immediate rise to many type features

<table>
<thead>
<tr>
<th>Feature Category</th>
<th>Feature Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear products</td>
<td>Separation Conjunction (*)</td>
</tr>
<tr>
<td>Function types</td>
<td>Wand (→*) and weakest precondition (wp e {Φ})</td>
</tr>
<tr>
<td>Session types</td>
<td>Actris dependent separation protocols (iProto)</td>
</tr>
<tr>
<td>Unique references</td>
<td>Points-to connective (ℓ ↦→ v)</td>
</tr>
<tr>
<td>Shared references</td>
<td>Invariants ([P])</td>
</tr>
<tr>
<td>Copyable types</td>
<td>Persistent modality (□)</td>
</tr>
<tr>
<td>Lock types</td>
<td>Iris’s lock library</td>
</tr>
<tr>
<td>Session choice types</td>
<td>Actris dependent separation protocols (iProto)</td>
</tr>
<tr>
<td>Recursion</td>
<td>Guarded step-indexed recursion (▷)</td>
</tr>
<tr>
<td>Term polymorphism</td>
<td>Higher-order impredicative quantifiers</td>
</tr>
<tr>
<td>Session polymorphism</td>
<td>Higher-order impredicative protocols binders</td>
</tr>
</tbody>
</table>
Overview of features

Iris and Actris gives immediate rise to many type features

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear products</td>
<td>Separation Conjunction (*)</td>
</tr>
<tr>
<td>Function types</td>
<td>Wand (→*) and weakest precondition (wp e {Φ})</td>
</tr>
<tr>
<td>Session types</td>
<td>Actris dependent separation protocols (iProto)</td>
</tr>
<tr>
<td>Unique references</td>
<td>Points-to connective (ℓ ↦→ v)</td>
</tr>
<tr>
<td>Shared references</td>
<td>Invariants ([P])</td>
</tr>
<tr>
<td>Copyable types</td>
<td>Persistent modality (□)</td>
</tr>
<tr>
<td>Lock types</td>
<td>Iris’s lock library</td>
</tr>
<tr>
<td>Session choice types</td>
<td>Actris dependent separation protocols (iProto)</td>
</tr>
<tr>
<td>Recursion</td>
<td>Guarded step-indexed recursion (⊿)</td>
</tr>
<tr>
<td>Term polymorphism</td>
<td>Higher-order impredicative quantifiers</td>
</tr>
<tr>
<td>Session polymorphism</td>
<td>Higher-order impredicative protocols binders</td>
</tr>
<tr>
<td>Term subtyping</td>
<td>Predicates closed under wand (∀v. A₁ v →* A₂ v)</td>
</tr>
</tbody>
</table>
### Overview of features

**Iris** and **Actris** gives immediate rise to many type features

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear products</td>
<td>Separation Conjunction (*)</td>
</tr>
<tr>
<td>Function types</td>
<td>Wand (→) and weakest precondition (wp e {Φ})</td>
</tr>
<tr>
<td>Session types</td>
<td>Actris dependent separation protocols (iProto)</td>
</tr>
<tr>
<td>Unique references</td>
<td>Points-to connective (ℓ ↦→ ν)</td>
</tr>
<tr>
<td>Shared references</td>
<td>Invariants ([P])</td>
</tr>
<tr>
<td>Copyable types</td>
<td>Persistent modality (□)</td>
</tr>
<tr>
<td>Lock types</td>
<td>Iris’s lock library</td>
</tr>
<tr>
<td>Session choice types</td>
<td>Actris dependent separation protocols (iProto)</td>
</tr>
<tr>
<td>Recursion</td>
<td>Guarded step-indexed recursion (▷)</td>
</tr>
<tr>
<td>Term polymorphism</td>
<td>Higher-order impredicative quantifiers</td>
</tr>
<tr>
<td>Session polymorphism</td>
<td>Higher-order impredicative protocols binders</td>
</tr>
<tr>
<td>Term subtyping</td>
<td>Predicates closed under wand (∀ν. A₁ ν →* A₂ ν)</td>
</tr>
<tr>
<td>Session subtyping</td>
<td>Actris 2.0 subprotocols (□)</td>
</tr>
</tbody>
</table>
Overview of features

Iris and Actris gives immediate rise to many type features

<table>
<thead>
<tr>
<th>Linear products</th>
<th>Separation Conjunction ((\ast))</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Subprotocols: \(prot_1 \sqsubseteq prot_2\)

- Generalisation of asynchronous subtyping for functional correctness
- Makes asynchronous semantics explicit by swap rule
  - \(?\langle v_1\rangle\{P_1\}.!\langle v_2\rangle\{P_2\}.prot \sqsubseteq !\langle v_2\rangle\{P_2\}.?\langle v_1\rangle\{P_1\}.prot\)
  - \(?A_1.\!A_2. S <\! A_2. ?A_1. S\)
- Non-trivial extension due to dependent binders and step-indexing
  - Required updates to the model of iProto

<table>
<thead>
<tr>
<th>Term polymorphism</th>
<th>Higher-order impredicative quantifiers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Session polymorphism</td>
<td>Higher-order impredicative protocols binders</td>
</tr>
<tr>
<td>Term subtyping</td>
<td>Predicates closed under wand ((\forall v. A_1 v \not\rightarrow A_2 v))</td>
</tr>
<tr>
<td>Session subtyping</td>
<td>Actris 2.0 subprotocols ((\sqsubseteq))</td>
</tr>
</tbody>
</table>
Overview of features - Definitions

Unique references: \(\text{ref}_{\text{uniq}} A \triangleq \lambda w. \exists v. w \in \text{Loc} \star (w \mapsto v) \star \triangledown (A v)\)

Shared references: \(\text{ref}_{\text{shr}} A \triangleq \lambda w. (w \in \text{Loc}) \star \exists v. (w \mapsto v) \star \Box (A v)\)

Copyable types: \(\text{copy} A \triangleq \lambda w. \Box (A w)\)

Lock types:
\(\text{mutex} A \triangleq \lambda w. \exists lk, \ell. (w = (lk, \ell)) \star \text{isLock} lk (\exists v. (\ell \mapsto u) \star \triangledown (A v))\)
\(\bar{\text{mutex}} A \triangleq \lambda w. \exists lk, \ell. (w = (lk, \ell)) \star \text{isLock} lk (\exists v. (\ell \mapsto u) \star \triangledown (A v)) \star (\ell \mapsto -)\)

Session choice:
\(\oplus \{\bar{S}\} \triangleq ! (l : \mathbb{Z}) \langle l \rangle \{\triangledown (l \in \text{dom}(\bar{S}))\}. \bar{S}(l)\)
\(& \{\bar{S}\} \triangleq ? (l : \mathbb{Z}) \langle l \rangle \{\triangledown (l \in \text{dom}(\bar{S}))\}. \bar{S}(l)\)

Recursion: \(\mu (X : k). K \triangleq \mu (X : \text{Type}_k). K \quad (K \text{ must be contractive in } X)\)

Polymorphism:
\(\forall (X : k). A \triangleq \lambda w. \forall (X : \text{Type}_k). \text{wp} w () \{A\}\)
\(\exists (X : k). A \triangleq \lambda w. \exists (X : \text{Type}_k). \triangledown (A w)\)
\(!_{\bar{X} : \bar{k}} A. S \triangleq ! (\bar{X} : \text{Type}_k) (v : \text{Val}) \langle v \rangle \{\triangledown (A v)\}. S\)
\(?_{\bar{X} : \bar{k}} A. S \triangleq ? (\bar{X} : \text{Type}_k) (v : \text{Val}) \langle v \rangle \{\triangledown (A v)\}. S\)

Term subtyping: \(A <: B \triangleq \forall v. A v \rightarrow B v\)

Session subtyping: \(S_1 <: S_2 \triangleq S_1 \sqsubseteq S_2\)
Typing the Untypeable
An Untypeable Program

Consider the following program:

\[ \lambda c. (\text{recv } c \mid\mid \text{recv } c) : \text{chan } (?Z. ?Z. \text{end}) \rightarrow (Z \times Z) \]
An Untypeable Program

Consider the following program:

$\equiv \lambda c. (\text{recv } c \parallel \text{recv } c) : \text{chan } (?Z. ?Z. \text{end}) \rightarrow (Z \times Z)$

Is it typeable?
Consider the following program:

\[ \lambda c. (\text{recv } c \parallel \text{recv } c) : \text{chan } (\text{?Z. ?Z. end}) \rightarrow (\text{Z } \times \text{Z}) \]

Is it typeable? No
Consider the following program:

\[ \lambda c. (\text{recv } c \mid\mid \text{recv } c) : \text{chan } (\text{?Z. ?Z. end}) \rightsquigarrow (\text{Z} \times \text{Z}) \]

Is it typeable?  No  It violates the ownership discipline
An Untypeable Program

Consider the following program:

$$\vdash \lambda c. (\text{recv } c \parallel \text{recv } c) : \text{chan } (?Z. ?Z. \text{end}) \to (Z \times Z)$$

Is it typeable?  No

It violates the ownership discipline

Is it safe?

Beyond the scope of this talk
An Untypeable Program

Consider the following program:

\[ \equiv \lambda c. (\text{recv } c \mid\mid \text{recv } c) : \text{chan } (?Z. ?Z. \text{end}) \xrightarrow{\circ} (Z \times Z) \]

Is it typeable?  No  It violates the ownership discipline
Is it safe?   Yes
An Untypeable Program

Consider the following program:

\[ \lambda c. (\text{recv } c \parallel \text{recv } c) : \text{chan} (\mathbf{?Z}. \mathbf{?Z}. \text{end}) \to (\mathbf{Z} \times \mathbf{Z}) \]

- Is it typeable? No  It violates the ownership discipline
- Is it safe? Yes  Order of receives does not matter
An Untypeable Program

Consider the following program:

\[ \lambda c. (\text{recv } c \mid \mid \text{recv } c) : \text{chan}(?Z. ?Z. \text{end}) \to (Z \times Z) \]

Is it typeable?  No  It violates the ownership discipline
Is it safe?  Yes  Order of receives does not matter
Really?
An Untypeable Program

Consider the following program:

\[ \lambda c. (\text{recv } c \parallel \text{recv } c) : \text{chan } (\text{?Z. ?Z. end}) \rightsquigarrow (\text{Z} \times \text{Z}) \]

Is it typeable? No It violates the ownership discipline
Is it safe? Yes Order of receives does not matter
Really? Well...
An Untypeable Program

Consider the following program:

$$\equiv \lambda c. (\text{recv } c \mid\mid \text{recv } c) : \text{chan } (\text{?Z. ?Z. end}) \rightarrow (\text{Z} \times \text{Z})$$

Is it typeable? No
Is it safe? Yes
Really? Well...

It violates the ownership discipline
Order of receives does not matter
It could be added as an ad-hoc rule

Beyond the scope of this talk
An Untypeable Program

Consider the following program:

$$
\iff \lambda c. (\text{recv } c \mid \text{recv } c) : \text{chan } (?Z. ?Z. \text{end}) \to (Z \times Z)
$$

Is it typeable? No  It violates the ownership discipline
Is it safe? Yes  Order of receives does not matter
Really? Well... It could be added as an ad-hoc rule

The rule is just another lemma
An Untypeable Program

Consider the following program:

$$\forall \lambda c. (\text{recv } c \parallel \text{recv } c) : \text{chan} (\text{?Z. ?Z. end}) \rightarrow (\text{Z } \times \text{Z})$$

Is it typeable? No It violates the ownership discipline
Is it safe? Yes Order of receives does not matter
Really? Well... It could be added as an ad-hoc rule

The rule is just another lemma proven by unfolding all type-level definitions

$$(c \mapsto ?(v_1 : \text{Val}) \langle v_1 \rangle \triangleright (v_1 \in \text{Z})). ?(v_2 : \text{Val}) \langle v_2 \rangle \triangleright (v_2 \in \text{Z})). \text{end} \rightarrow$$

wp (recv c || recv c) {v. \exists v_1, v_2. (v = (v_1, v_2)) \triangleright (v_1 \in \text{Z}) \triangleright (v_2 \in \text{Z})}$$
An Untypeable Program

Consider the following program:

$$\vdash \lambda c. (\text{recv } c \parallel \text{recv } c) : \text{chan } (?\mathbb{Z}. ?\mathbb{Z}. \text{end}) \rightarrow (\mathbb{Z} \times \mathbb{Z})$$

Is it typeable?  No It violates the ownership discipline
Is it safe?  Yes Order of receives does not matter
Really?  Well... It could be added as an ad-hoc rule

The rule is just another lemma proven by unfolding all type-level definitions

$$(c \mapsto ?(v_1 : \text{Val}) \langle v_1 \rangle \{\triangleright (v_1 \in \mathbb{Z})\}. ?(v_2 : \text{Val}) \langle v_2 \rangle \{\triangleright (v_2 \in \mathbb{Z})\}. \text{end}) \rightarrow$$

wp (\text{recv } c \parallel \text{recv } c) \{v. \exists v_1, v_2. (v = (v_1, v_2)) \triangleright (v_1 \in \mathbb{Z}) \triangleright (v_2 \in \mathbb{Z})\}

And then using Iris’s ghost state machinery!
An Untypeable Program

Consider the following program:

\[ \lambda c. (\text{recv } c \parallel \text{recv } c) : \text{chan } (\texttt{?Z. }\texttt{?Z. }\texttt{end}) \rightarrow (\texttt{Z }\times\texttt{Z}) \]

Is it typeable? No
It violates the ownership discipline
Is it safe? Yes
Order of receives does not matter
Really? Well...
It could be added as an ad-hoc rule

The rule is just another lemma proven by unfolding all type-level definitions

\[
(c \rightarrow ?(v_1 : \text{Val}) \langle v_1 \rangle \{\triangleright (v_1 \in \texttt{Z})\}. ?(v_2 : \text{Val}) \langle v_2 \rangle \{\triangleright (v_2 \in \texttt{Z})\}. \texttt{end}) \rightarrow^* \wp (\text{recv } c \parallel \text{recv } c) \{v. \exists v_1, v_2. \((v = (v_1, v_2)) \triangleright (v_1 \in \texttt{Z}) \triangleright (v_2 \in \texttt{Z})\} \]

And then using Iris's ghost state machinery! Beyond the scope of this talk
Concluding Remarks
Concluding Remarks

Semantic typing and separation logic is a good fit for mechanising session types

- **Linearity** is implicit from separation logic
- **Binders** can be inherited from underlying logic

Using a strong logic gives immediate rise to advanced features

- **Iris**: Polymorphism, recursion, locks and more
- **Actris**: Session types, session polymorphism, session subtyping

Sources:

- Paper ([https://iris-project.org/pdfs/2020-actris2-submission.pdf](https://iris-project.org/pdfs/2020-actris2-submission.pdf))
Questions?
Subtyping
Semantic Asynchronous Session Subtyping

Conventional subtyping:

\[
\begin{align*}
S_1 & \ll S_2 \\
\text{chan } S_1 & \ll \text{chan } S_2 \\
A_2 & \ll A_1 & S_1 & \ll S_2 \\
!A_1. S_1 & \ll !A_2. S_2 \\
?A_1. S_1 & \ll ?A_2. S_2
\end{align*}
\]
Semantic Asynchronous Session Subtyping

Conventional subtyping:

\[
\begin{align*}
S_1 & \ll S_2 \\
\text{chan } S_1 & \ll \text{chan } S_2
\end{align*}
\]

\[
\begin{align*}
A_2 & \ll A_1 \\
S_1 & \ll S_2 \\
!A_1. S_1 & \ll !A_2. S_2
\end{align*}
\]

\[
\begin{align*}
A_1 & \ll A_2 \\
S_1 & \ll S_2 \\
?A_1. S_1 & \ll ?A_2. S_2
\end{align*}
\]

Asynchronous Subtyping:

\[
\]
Semantic Asynchronous Session Subtyping

**Conventional subtyping:**

\[
\begin{align*}
S_1 & <: S_2 \\
\text{chan } S_1 & <: \text{chan } S_2 \\
A_2 & <: A_1 & S_1 & <: S_2 \\
!A_1. S_1 & <: !A_2. S_2 \\
A_1 & <: A_2 & S_1 & <: S_2 \\
\end{align*}
\]

**Asynchronous Subtyping:**

\[
\]

**Polymorphism subtyping:**

\[
\begin{align*}
!_{(\vec{x}:\vec{k})} A. S & <: !A[\vec{K}/\vec{X}]. S[\vec{K}/\vec{X}] \\
?A[\vec{K}/\vec{X}]. S[\vec{K}/\vec{X}] & <: ?_{(\vec{x}:\vec{k})} A. S \\
S_1 & <: !A. S_2 \\
S_1 & <: !_{(\vec{x}:\vec{k})} A. S_2 \\
?A. S_1 & <: S_2 \\
?_{(\vec{x}:\vec{k})} A. S_1 & <: S_2
\end{align*}
\]
Semantic Asynchronous Session Subtyping - Example

Goal:
\[
\mu (\text{rec} : \Diamond). !(X, Y: \star) (X \to Y). \! X. \? Y. \text{rec} <: \mu (\text{rec} : \Diamond). !(X_1, X_2: \star) (X_1 \to B). \! X_1. !(X_2 \to Z). \! X_2. \? B. \? Z. \text{rec}
\]
Semantic Asynchronous Session Subtyping - Example

Goal:
\[ \mu \left( \text{rec : } \diamond \right) \cdot (X,Y : \star) (X \rightarrow Y) . !X . ?Y . \text{rec} <: \mu \left( \text{rec : } \diamond \right) . !(X_1,X_2 : \star) (X_1 \rightarrow B) . !X_1 . !(X_2 \rightarrow Z) . !X_2 . ?B . ?Z . \text{rec} \]

Derivation:
\[ \mu \left( \text{rec : } \diamond \right) \cdot (X,Y : \star) (X \rightarrow Y) . !X . ?Y . \text{rec} \]
Semantic Asynchronous Session Subtyping - Example

Goal:

\[ \mu (\text{rec} : \diamond). !(x, y : \star) (X \rightarrow Y). !X. ?Y. \text{rec} \lessdot \mu (\text{rec} : \diamond). !(x_1, x_2 : \star) (X_1 \rightarrow B). !X_1. !(X_2 \rightarrow Z). !X_2. ?B. ?Z. \text{rec} \]

Derivation:

\[ \mu (\text{rec} : \diamond). !(x, y : \star) (X \rightarrow Y). !X. ?Y. \text{rec} \]
\[ \lessdot \mu (\text{rec} : \diamond). !(x_1, y_1 : \star) (X_1 \rightarrow Y_1). !X_1. ?Y_1. !(x_2, y_2 : \star) (X_2 \rightarrow Y_2). !X_2. ?Y_2. \text{rec} \]

(LÖB)
Semantic Asynchronous Session Subtyping - Example

Goal:
\[ \mu (\text{rec} : \diamond) . ! (X, Y : \star) (X \rightarrow Y) . ! X . ? Y . \text{rec} <: \mu (\text{rec} : \diamond) . ! (X_1, X_2 : \star) (X_1 \rightarrow B) . ! X_1 . !(X_2 \rightarrow Z) . ! X_2 . ? B . ? Z . \text{rec} \]

Derivation:
\[ \mu (\text{rec} : \diamond) . ! (X, Y : \star) (X \rightarrow Y) . ! X . ? Y . \text{rec} <: \mu (\text{rec} : \diamond) . ! (X_1, Y_1 : \star) (X_1 \rightarrow Y_1) . ! X_1 . ? Y_1 . ! (X_2, Y_2 : \star) (X_2 \rightarrow Y_2) . ! X_2 . ? Y_2 . \text{rec} \]
\[ <: \mu (\text{rec} : \diamond) . ! (X_1, X_2 : \star) (X_1 \rightarrow B) . ! X_1 . ? B . !(X_2 \rightarrow Z) . ! X_2 . ? Z . \text{rec} \]  
\[ (\text{LÖB}) \]
\[ <: \mu (\text{rec} : \diamond) . ! (X_1, X_2 : \star) (X_1 \rightarrow B) . ! X_1 . ? B . !(X_2 \rightarrow Z) . ! X_2 . ? Z . \text{rec} \]  
\[ (\text{S-ELIM, S-INTRO}) \]

Rules:
\[
\text{S-ELIM} \\
\frac{S_1 <: ! A . S_2}{S_1 <: ! (\vec{X} : \vec{k}) A . S_2} \\
\text{S-INTRO} \\
\frac{! (\vec{X} : \vec{k}) A . S <: ! A[\vec{K} / \vec{X}] . S[\vec{K} / \vec{X}]}{! A . S} 
\]
Semantic Asynchronous Session Subtyping - Example

Goal:

\[
\mu (\text{rec} : \diamond). !(X, Y : \star) (X \to Y). !X. ?Y. \text{rec} <: \mu (\text{rec} : \diamond). !(x_1, x_2 : \star) (x_1 \to B). !x_1. !(x_2 \to Z). !x_2. ?B. ?Z. \text{rec}
\]

Derivation:

\[
\mu (\text{rec} : \diamond). !(X, Y : \star) (X \to Y). !X. ?Y. \text{rec} \\
<: \mu (\text{rec} : \diamond). !(x_1, y_1 : \star) (x_1 \to y_1). !x_1. ?y_1. !(x_2, y_2 : \star) (x_2 \to y_2). !x_2. ?y_2. \text{rec} \\
<: \mu (\text{rec} : \diamond). !(x_1, x_2 : \star) (x_1 \to B). !x_1. ?B. !(x_2 \to Z). !x_2. ?Z. \text{rec} \\
<: \mu (\text{rec} : \diamond). !(x_1, x_2 : \star) (x_1 \to B). !x_1. !(x_2 \to Z). ?B. !x_2. ?Z. \text{rec}
\]

(Rules):

\[
\begin{align*}
\text{S-ELIM} & \\
S_1 <: !A. S_2 & \\
\hline
S_1 <: !(\vec{X} : \vec{k}) A. S_2 & \\
\text{S-INTRO} & \\
!_{(\vec{X} : \vec{k})} A. S <: !A[\vec{k} / \vec{X}]. S[\vec{k} / \vec{X}] & \\
\text{SWAP} & \\
\end{align*}
\]
Semantic Asynchronous Session Subtyping - Example

Goal:

\[ \mu (rec : \diamondsuit). !((X \to Y) \to X. ?Y. rec <: \mu (rec : \diamondsuit). !(X_1 \to B) \to X_1. !(X_2 \to Z). !X_2. ?B. ?Z. rec \]

Derivation:

\[ \mu (rec : \diamondsuit). !((X \to Y) \to X. ?Y. rec <: \mu (rec : \diamondsuit). !(X_1 \to Y_1) \to X_1. ?Y_1. !(X_2 \to Y_2) \to X_2. ?Y_2. rec \]

\[ <: \mu (rec : \diamondsuit). !(X_1, X_2 : \star) \to X_1. ?B. !(X_2 \to Z). !X_2. ?Z. rec \]

(S-ELIM, S-INTRO)

\[ <: \mu (rec : \diamondsuit). !(X_1, X_2 : \star) \to X_1. !(X_2 \to Z). ?B. !X_2. ?Z. rec \]

(SWAP)

\[ <: \mu (rec : \diamondsuit). !(X_1, X_2 : \star) \to X_1. !(X_2 \to Z). !X_2. ?B. ?Z. rec \]

(SWAP)

Rules:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Premises</th>
<th>Conclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>S-ELIM</td>
<td>[ S_1 \vdash \lnot A. S_2 ]</td>
<td>[ S_1 \vdash \lnot (X : \star) A. S_2 ]</td>
</tr>
<tr>
<td>S-INTRO</td>
<td>[ !((X_1 \to Y_1) \to X_1. ?Y_1. !(X_2 \to Y_2) \to X_2. ?Y_2. rec ]</td>
<td>[ !((X_1 \to Y_1) \to X_1. ?Y_1. !(X_2 \to Y_2) \to X_2. ?Y_2. rec ]</td>
</tr>
<tr>
<td>SWAP</td>
<td>[ ?A_1. !A_2. S \vdash \lnot A_1. !A_2. S ]</td>
<td>[ ?A_1. !A_2. S \vdash \lnot A_1. !A_2. S ]</td>
</tr>
</tbody>
</table>