Machine-Checked Semantic Session Typing

Jonas Kastberg Hinrichsen, IT University of Copenhagen

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

20. October 2020
IT University of Copenhagen

Problem

Mechanising type systems is hard

Problem

Mechanising type systems is hard

- **Binders** impose non-trivial proof effort

Problem

Mechanising type systems is hard

- **Binders** impose non-trivial proof effort
- **Substructural Properties** requires explicit handling

Problem

Mechanising type systems is hard

- **Binders** impose non-trivial proof effort
- **Substructural Properties** requires explicit handling
- **Extensions** impose immodular proof effort

Problem

Mechanising type systems is hard, especially **syntactic type systems**

- **Binders** impose non-trivial proof effort
- **Substructural Properties** requires explicit handling
- **Extensions** impose immodular proof effort

What is a Type System?

"A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute." - Benjamin Pierce, Types and Programming Languages

What is a Type System?

“A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute.” - Benjamin Pierce, Types and Programming Languages

- **Terms:** Program phrases

What is a Type System?

“A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute.” - Benjamin Pierce, Types and Programming Languages

- **Terms:** Program phrases
- **Types:** Kinds of values

What is a Type System?

“A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute.” - Benjamin Pierce, Types and Programming Languages

- **Terms:** Program phrases
- **Types:** Kinds of values
- **Rules:** Relations between **Terms** and **Types**

What is a Type System?

“A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute.” - Benjamin Pierce, Types and Programming Languages

- **Terms:** Program phrases
- **Types:** Kinds of values
- **Rules:** Relations between **Terms** and **Types**
- **Soundness:** The absence of certain behaviours
    - *Safety:* Absence of crashes
    - *Deadlock-Freedom:* Absence of waiting indefinitely

Syntactic Typing

In a **syntactic type system**

Syntactic Typing

In a **syntactic type system**

- **Types** are defined as a closed inductive definition

Syntactic Typing

In a **syntactic type system**

- **Types** are defined as a closed inductive definition: $\tau ::= \mathbb{Z} \mid \mathbb{B} \mid \ldots$

Syntactic Typing

In a **syntactic type system**

- **Types** are defined as a closed inductive definition: $\tau ::= \mathbb{Z} \mid \mathbb{B} \mid \ldots$
- **Rules** are defined as a closed inductive relation

Syntactic Typing

In a **syntactic type system**

- **Types** are defined as a closed inductive definition: $\tau ::= \mathbb{Z} \mid \mathbb{B} \mid \ldots$
- **Rules** are defined as a closed inductive relation: $\vdash i : \mathbb{Z}$

Syntactic Typing

In a **syntactic type system**

- **Types** are defined as a closed inductive definition: $\tau ::= \mathbb{Z} \mid \mathbb{B} \mid \ldots$
- **Rules** are defined as a closed inductive relation: $\vdash i : \mathbb{Z}$
- **Soundness** is proven as **progress**/ **preservation**

Syntactic Typing

In a **syntactic type system**

- **Types** are defined as a closed inductive definition: $\tau ::= \mathbb{Z} \mid \mathbb{B} \mid \ldots$
- **Rules** are defined as a closed inductive relation: $\vdash i : \mathbb{Z}$
- **Soundness** is proven as **progress**/ **preservation** using induction on the relation

Syntactic Typing

In a **syntactic type system**

- **Types** are defined as a closed inductive definition: $\tau ::= \mathbb{Z} \mid \mathbb{B} \mid \ldots$
- **Rules** are defined as a closed inductive relation: $\vdash i : \mathbb{Z}$
- **Soundness** is proven as **progress**/ **preservation** using induction on the relation

Binders impose non-trivial proof effort

- Manual capture-avoiding substitution/renaming

Syntactic Typing

In a **syntactic type system**

- **Types** are defined as a closed inductive definition: $\tau ::= \mathbb{Z} \mid \mathbb{B} \mid \ldots$
- **Rules** are defined as a closed inductive relation: $\vdash i : \mathbb{Z}$
- **Soundness** is proven as **progress**/ **preservation** using induction on the relation

Binders impose non-trivial proof effort

- Manual capture-avoiding substitution/renaming

Substructural Properties requires explicit handling

- Explicit context splitting in rules (for linearity)

Syntactic Typing

In a **syntactic type system**

- **Types** are defined as a closed inductive definition: $\tau ::= \mathbb{Z} \mid \mathbb{B} \mid \ldots$
- **Rules** are defined as a closed inductive relation: $\vdash i : \mathbb{Z}$
- **Soundness** is proven as **progress**/ **preservation** using induction on the relation

Binders impose non-trivial proof effort

- Manual capture-avoiding substitution/renaming

Substructural Properties requires explicit handling

- Explicit context splitting in rules (for linearity)

Extensions impose immodular proof effort

- Must reprove **progress** and **preservation** when adding types/rules

Syntactic Typing

In a **syntactic type system**

- **Types** are defined as a closed inductive definition: $\tau ::= \mathbb{Z} \mid \mathbb{B} \mid \ldots$
- **Rules** are defined as a closed inductive relation: $\vdash i : \mathbb{Z}$
- **Soundness** is proven as **progress**/ **preservation** using induction on the relation

Binders impose non-trivial proof effort

- Manual capture-avoiding substitution/renaming

Substructural Properties requires explicit handling

- Explicit context splitting in rules (for linearity)

Extensions impose immodular proof effort

- Must reprove **progress** and **preservation** when adding types/rules
- Adding unsound rules makes entire type system unsound

Syntactic Typing

In a **syntactic type system**

- **Types** are defined as a closed inductive definition: $\tau ::= \mathbb{Z} \mid \mathbb{B} \mid \ldots$
- **Rules** are defined as a closed inductive relation: $\vdash i : \mathbb{Z}$
- **Soundness** is proven as **progress**/ **preservation** using induction on the relation

Binders impose non-trivial proof effort

- Manual capture-avoiding substitution/renaming

Substructural Properties requires explicit handling

- Explicit context splitting in rules (for linearity)

Extensions impose immodular proof effort

- Must reprove **progress** and **preservation** when adding types/rules
- Adding unsound rules makes entire type system unsound: $\vdash b : \mathbb{Z}$ ✗

Goal:

A “mechanisable” type system

Solution:

A semantic type system!

Solution - Semantic Typing!

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

Solution - Semantic Typing!

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

- **Types** defined as predicates over values

Solution - Semantic Typing!

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

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

Solution - Semantic Typing!

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

- **Types** defined as predicates over values, e.g.: $\mathsf{Z} \triangleq \lambda w.\, w \in \mathbb{Z}$
- **Judgement** defined as safety-capturing evaluation: $\Gamma \vDash e : A$

Solution - Semantic Typing!

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

- **Types** defined as predicates over values, e.g.: $\mathsf{Z} \triangleq \lambda w.\, w \in \mathbb{Z}$
- **Judgement** defined as safety-capturing evaluation: $\Gamma \vDash e : A$

  e does not get *stuck*

Solution - Semantic Typing!

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

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

Solution - Semantic Typing!

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

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

Solution - Semantic Typing!

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

- **Types** defined as predicates over values, e.g.: $\mathsf{Z} \triangleq \lambda\, w.\, w \in \mathbb{Z}$
- **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: $\vDash i : \mathsf{Z}$

Solution - Semantic Typing!

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

- **Types** defined as predicates over values, e.g.: $\mathsf{Z} \triangleq \lambda\, w.\, w \in \mathbb{Z}$
- **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: $\vDash i : \mathsf{Z} \quad \rightsquigarrow \quad i \in \mathbb{Z}$

Solution - Semantic Typing!

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

- **Types** defined as predicates over values, e.g.: $\mathsf{Z} \triangleq \lambda\, w.\, w \in \mathbb{Z}$
- **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: $\vDash i : \mathsf{Z} \quad \rightsquigarrow \quad i \in \mathbb{Z}$
- **Soundness** is a consequence of the judgement definition

Solution - Semantic Typing!

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

- **Types** defined as predicates over values, e.g.: $\mathsf{Z} \triangleq \lambda\, w.\, w \in \mathbb{Z}$
- **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: $\vDash i : \mathsf{Z} \quad \rightsquigarrow \quad i \in \mathbb{Z}$
- **Soundness** is a consequence of the judgement definition

Handling of **substructural properties** and **binders** can be inherited from the logic

Solution - Semantic Typing!

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

- **Types** defined as predicates over values, e.g.: $\mathsf{Z} \triangleq \lambda\, w.\, w \in \mathbb{Z}$
- **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: $\vDash i : \mathsf{Z} \quad \rightsquigarrow \quad i \in \mathbb{Z}$
- **Soundness** is a consequence of the judgement definition

Handling of **substructural properties** and **binders** can be inherited from the logic

Extensions can be added modularly

Solution - Semantic Typing!

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

- **Types** defined as predicates over values, e.g.: $\mathrm{Z} \triangleq \lambda\, w.\, w \in \mathbb{Z}$
- **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: $\vDash i : \mathrm{Z} \quad \rightsquigarrow \quad i \in \mathbb{Z}$
- **Soundness** is a consequence of the judgement definition

Handling of **substructural properties** and **binders** can be inherited from the 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, e.g.: $\mathsf{Z} \triangleq \lambda\, w.\, w \in \mathbb{Z}$
- **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: $\vDash i : \mathsf{Z} \quad \rightsquigarrow \quad i \in \mathbb{Z}$
- **Soundness** is a consequence of the judgement definition

Handling of **substructural properties** and **binders** can be inherited from the logic

Extensions can be added modularly

- Adding types and rules does not inherently impose new proof effort on existing types, rules and soundness: $\mathsf{B} \triangleq \lambda\, w.\, w \in \mathbb{B}$

Solution - Semantic Typing!

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

- **Types** defined as predicates over values, e.g.: $\mathsf{Z} \triangleq \lambda\, w.\, w \in \mathbb{Z}$
- **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: $\vDash i : \mathsf{Z} \quad \rightsquigarrow \quad i \in \mathbb{Z}$
- **Soundness** is a consequence of the judgement definition

Handling of **substructural properties** and **binders** can be inherited from the logic

Extensions can be added modularly

- Adding types and rules does not inherently impose new proof effort on existing types, rules and soundness: $\mathsf{B} \triangleq \lambda\, w.\, w \in \mathbb{B} \qquad \vDash b : \mathsf{B}$

Solution - Semantic Typing!

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

- **Types** defined as predicates over values, e.g.: $\mathrm{Z} \triangleq \lambda\, w.\, w \in \mathbb{Z}$
- **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: $\vDash i : \mathrm{Z} \quad \rightsquigarrow \quad i \in \mathbb{Z}$
- **Soundness** is a consequence of the judgement definition

Handling of **substructural properties** and **binders** can be inherited from the logic

Extensions can be added modularly

- Adding types and rules does not inherently impose new proof effort on existing types, rules and soundness: $\mathrm{B} \triangleq \lambda\, w.\, w \in \mathbb{B} \qquad \vDash b : \mathrm{B}$
- Unsound rules cannot be added (proven)

Solution - Semantic Typing!

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

- **Types** defined as predicates over values, e.g.: $\mathsf{Z} \triangleq \lambda\, w.\, w \in \mathbb{Z}$
- **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: $\vDash i : \mathsf{Z} \quad \rightsquigarrow \quad i \in \mathbb{Z}$
- **Soundness** is a consequence of the judgement definition

Handling of **substructural properties** and **binders** can be inherited from the logic

Extensions can be added modularly

- Adding types and rules does not inherently impose new proof effort on existing types, rules and soundness: $\mathsf{B} \triangleq \lambda\, w.\, w \in \mathbb{B} \qquad \vDash b : \mathsf{B}$
- Unsound rules cannot be added (proven): $\vDash b : \mathsf{Z}$ ✗

Case study:
Semantic Session Type System

Semantic Typing

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

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

Semantic Typing using **Iris**

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

- **Substructural properties** 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)

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

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

- **Substructural properties** 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**

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/-/tree/cpp21)
- Supports integrating safe yet untypeable programs

Semantic Session Type System

Language

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

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

Language

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

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

Only allows substitution with closed terms

- To avoid substitution overhead

Language

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

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

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. $\texttt{send}\ c\ (\texttt{recv}\ c)$)

Language

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

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

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. `send` c (`recv` c))

Message-passing is:

- **Binary:** Each channel have one pair of endpoints
- **Asynchronous:** `send` does not block, two buffers per endpoint pair
- **Affine:** No `close` expression, channels can be thrown away

Semantic Term Types

Types as Iris predicates:

$$\mathsf{Type}_\bigstar \triangleq \mathsf{Val} \to \mathsf{iProp}$$

Semantic Term Types

Types as Iris predicates:

$$\mathsf{Type}_\bigstar \triangleq \mathsf{Val} \to \mathsf{iProp}$$
$$\mathsf{Z} \triangleq \lambda\, w.\, w \in \mathbb{Z}$$

Semantic Term Types

Types as Iris predicates:

$$\begin{aligned} \mathsf{Type}_\bigstar &\triangleq \mathsf{Val} \to \mathsf{iProp} \\ \mathsf{Z} &\triangleq \lambda\, w.\, w \in \mathbb{Z} \\ A_1 \times A_2 &\triangleq \lambda\, w.\, \exists w_1, w_2.\, w = (w_1, w_2) * \triangleright(A_1\, w_1) * \triangleright(A_2\, w_2) \end{aligned}$$

Semantic Term Types

Types as Iris predicates:

$$
\begin{aligned}
\mathsf{Type}_{\bigstar} &\triangleq \mathsf{Val} \to \mathsf{iProp} \\
\mathsf{Z} &\triangleq \lambda\, w.\, w \in \mathbb{Z} \\
A_1 \times A_2 &\triangleq \lambda\, w.\, \exists w_1, w_2.\, w = (w_1, w_2) * \triangleright(A_1\, w_1) * \triangleright(A_2\, w_2) \\
\texttt{ref}_{\texttt{uniq}}\, A &\triangleq \lambda\, w.\, \exists v.\, w \in \mathsf{Loc} * (w \mapsto v) * \triangleright(A\, v)
\end{aligned}
$$

Semantic Term Types

Types as Iris predicates:

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

wp e $\{v.\Phi\}$ dictates e does not get *stuck* and if e reduces to a value v then Φ v holds

Semantic Term Types

Types as Iris predicates:

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

Judgement

$$\Gamma \vDash e : A \dashv \Gamma'$$

$\mathsf{wp}\ e\ \{v.\Phi\}$ 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:

$$\begin{aligned}
\mathsf{Type}_\bigstar &\triangleq \mathsf{Val} \to \mathsf{iProp} \\
\mathsf{Z} &\triangleq \lambda\, w.\, w \in \mathbb{Z} \\
A_1 \times A_2 &\triangleq \lambda\, w.\, \exists w_1, w_2.\, w = (w_1, w_2) * \triangleright(A_1\, w_1) * \triangleright(A_2\, w_2) \\
\texttt{ref}_{\texttt{uniq}}\, A &\triangleq \lambda\, w.\, \exists v.\, w \in \mathsf{Loc} * (w \mapsto v) * \triangleright(A\, v) \\
A \multimap B &\triangleq \lambda\, w.\, \forall v.\, \triangleright(A\, v) \mathrel{-\!\!*} \mathsf{wp}\, (w\ v)\, \{B\}
\end{aligned}$$

Judgement as Iris weakest precondition:

$$\Gamma \vDash e : A \dashv \Gamma' \triangleq \forall \sigma.\, (\Gamma \vDash \sigma) \mathrel{-\!\!*} \mathsf{wp}\; e[\sigma]\, \{v. A\, v * (\Gamma' \vDash \sigma)\}$$

$\mathsf{wp}\; e\, \{v.\Phi\}$ 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:

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

Judgement as Iris weakest precondition:

$$\Gamma \vDash e : A \dashv \Gamma' \triangleq \forall \sigma.\, (\Gamma \vDash \sigma) \mathrel{-\!\!*} \mathsf{wp}\ e[\sigma]\,\{v. A\, v * (\Gamma' \vDash \sigma)\}$$

Soundness: If $[\,] \vDash e : A \dashv \Gamma$ then e does not get stuck

- Consequence of Iris's adequacy of weakest precondition

$\mathsf{wp}\ e\,\{v.\Phi\}$ dictates e does not get *stuck* and if e reduces to a value v then $\Phi\ v$ holds

Semantic Term Types - Rules

Rules:

$$\Gamma \vDash i : \mathsf{Z}$$

$$\frac{\Gamma_2 \vDash e_1 : A_1 \dashv \Gamma_3 \qquad \Gamma_1 \vDash e_2 : A_2 \dashv \Gamma_2}{\Gamma_1 \vDash (e_1, e_2) : A_1 \times A_2 \dashv \Gamma_3}$$

If $[\,] \vDash e : A \dashv \Gamma$
then e does not get stuck

Semantic Term Types - Rules

Rules:

$$\Gamma \vDash i : \mathbb{Z}$$

$$\frac{\Gamma_2 \vDash e_1 : A_1 \dashv \Gamma_3 \qquad \Gamma_1 \vDash e_2 : A_2 \dashv \Gamma_2}{\Gamma_1 \vDash (e_1, e_2) : A_1 \times A_2 \dashv \Gamma_3}$$

If $[\,] \vDash e : A \dashv \Gamma$
then e does not get stuck

Proofs:

```
Lemma ltyped_int Γ (i : Z) : ⊢ Γ ⊨ #i : lty_int.
Proof. iIntros "!>" (vs) "Henv /=". iApply wp_value. eauto. Qed.
```

```
Lemma ltyped_pair Γ1 Γ2 Γ3 e1 e2 A1 A2 :
  (Γ2 ⊨ e1 : A1 ⫤ Γ3) -∗ (Γ1 ⊨ e2 : A2 ⫤ Γ2) -∗
  Γ1 ⊨ (e1,e2) : A1 * A2 ⫤ Γ3.
Proof.
  iIntros "#H1 #H2". iIntros (vs) "!> HΓ /=".
  wp_apply (wp_wand with "(H2 [HΓ //])"); iIntros (w2) "[HA2 HΓ]".
  wp_apply (wp_wand with "(H1 [HΓ //])"); iIntros (w1) "[HA1 HΓ]".
  wp_pures. iFrame "HΓ". iExists w1, w2. by iFrame.
Qed.
```

```
Lemma ltyped_safety `{heapPreG Σ} e σ es σ' e' :
  (∀ `{heapG Σ}, ∃ A Γ', ⊢ ∅ ⊨ e : A ⫤ Γ') →
  rtc erased_step ([e], σ) (es, σ') → e' ∈ es →
  is_Some (to_val e') ∨ reducible e' σ'.
Proof.
  intros Hty. apply (heap_adequacy Σ NotStuck e σ (λ _, True))=> // ?.
  destruct (Hty _) as (A & Γ' & He). iIntros "_".
  iDestruct (He $!∅ with "[]") as "He"; first by rewrite /env_ltyped.
  iEval (rewrite -(subst_map_empty e)). iApply (wp_wand with "He"); auto.
Qed.
```

But what about session types?

Semantic Session Types - Definitions

Session types as a new type kind:

$$\begin{aligned} \text{Type}_{\blacklozenge} &\triangleq ? \\ \mathbf{!}A.\,S &\triangleq ? \\ \mathbf{?}A.\,S &\triangleq ? \\ \texttt{end} &\triangleq ? \end{aligned} \qquad \begin{aligned} \text{Type}_{\bigstar} &\triangleq \text{Val} \to \text{iProp} \\ \texttt{chan}\ S &\triangleq \lambda\, w.\, ? \end{aligned}$$

Requires capturing:

- **Linearity** of channel endpoint ownership
- **Delegation** of linear types / channels
- **Session fidelity** of communicated messages

Actris Dependent Separation Protocols

Session type-inspired protocols for functional correctness

	Dependent separation protocols	**Syntactic session types**
Example	$\mathbf{?}\,(x:\mathbb{Z})\,\langle x\rangle\{x > 10\}.\,\mathbf{?}\,\langle x+10\rangle\{\mathsf{True}\}.\,\mathsf{end}$	$\mathbf{?}\mathsf{Z}.\,\mathbf{?}\mathsf{Z}.\,\mathtt{end}$
Usage	$c \rightarrowtail \mathit{prot}$	$c\ :\ \mathtt{chan}\ S$

Semantic Session Types - Definitions

Session types as dependent separation protocols:

$$\begin{aligned} \mathsf{Type}_{\blacklozenge} &\triangleq \mathsf{iProto} \\ \mathbf{!}A.\,S &\triangleq \mathbf{!}\,(v : \mathsf{Val})\,\langle v\rangle\{A\,v\}.\,S \\ \mathbf{?}A.\,S &\triangleq \mathbf{?}\,(v : \mathsf{Val})\,\langle v\rangle\{A\,v\}.\,S \\ \mathtt{end} &\triangleq \mathsf{end} \end{aligned} \qquad \begin{aligned} \mathsf{Type}_{\bigstar} &\triangleq \mathsf{Val} \to \mathsf{iProp} \\ \mathtt{chan}\ S &\triangleq \lambda\, w.\, w \rightarrowtail S \end{aligned}$$

Dependent separation protocols:

Example: $\mathbf{?}\,(x : \mathbb{Z})\,\langle x\rangle\{x > 10\}.\,\mathbf{?}\,\langle x + 10\rangle\{\mathsf{True}\}.\,\mathsf{end}$

Usage: $c \rightarrowtail \mathit{prot}$

Semantic Session Types - Rules

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

$$
\begin{aligned}
\Gamma &\vDash \texttt{new\_chan}\ () : \texttt{chan}\ S \times \texttt{chan}\ \overline{S} \dashv \Gamma \\
\Gamma, (c : \texttt{chan}\ (\mathbf{!}A.\, S)), (x : A) &\vDash \texttt{send}\ c\ x : 1 \dashv \Gamma, (c : \texttt{chan}\ S) \\
\Gamma, (c : \texttt{chan}\ (\mathbf{?}A.\, S)) &\vDash \texttt{recv}\ c : A \dashv \Gamma, (c : \texttt{chan}\ S)
\end{aligned}
$$

Rule:

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

Proof:

```
Lemma ltyped_recv Γ (x : string) A S :
  Γ !! x = Some (chan (<??> TY A; S))%lty →
  ⊢ Γ ⊨ recv x : A ⫤ <[x:=(chan S)%lty]> Γ.
Proof.
  iIntros (Hx) "!>". iIntros (vs) "HΓ"=> /=.
  iDestruct (env_ltyped_lookup _ _ _ _ (Hx) with "HΓ") as (v' Heq) "[Hc HΓ]".
  rewrite Heq.
  wp_recv (v) as "HA". iFrame "HA".
  iDestruct (env_ltyped_insert _ _ x (chan _) _ with "[Hc //] HΓ") as "HΓ"=> /=.
  by rewrite insert_delete (insert_id vs).
Qed.
```

Extensions

Overview of features

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

Overview of features

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

Linear products	Separation Conjunction ($*$)

Overview of features

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

Linear products	Separation Conjunction ($*$)
Function types	Wand ($-\!*$) and weakest precondition ($\mathsf{wp}\ e\ \{\Phi\}$)

Overview of features

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

Linear products	Separation Conjunction ($*$)
Function types	Wand ($-\!*$) and weakest precondition ($\text{wp}\ e\ \{\Phi\}$)
Session types	Actris dependent separation protocols (iProto)

Overview of features

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

Linear products	Separation Conjunction ($*$)
Function types	Wand ($-\!\!*$) and weakest precondition ($\mathrm{wp}\ e\ \{\Phi\}$)
Session types	Actris dependent separation protocols (iProto)
Unique references	Points-to connective ($\ell \mapsto v$)

Overview of features

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

Linear products	Separation Conjunction ($*$)
Function types	Wand ($-\!*$) and weakest precondition ($\mathsf{wp}\ e\ \{\Phi\}$)
Session types	Actris dependent separation protocols (iProto)
Unique references	Points-to connective ($\ell \mapsto v$)
Shared references	Invariants ($\boxed{P}$)

Overview of features

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

Linear products	Separation Conjunction ($*$)
Function types	Wand ($-\!\!*$) and weakest precondition ($\mathsf{wp}\ e\ \{\Phi\}$)
Session types	Actris dependent separation protocols (iProto)
Unique references	Points-to connective ($\ell \mapsto v$)
Shared references	Invariants ($\boxed{P}$)
Copyable types	Persistent modality ($\square\, P$)

Overview of features

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

Linear products	Separation Conjunction ($*$)
Function types	Wand ($-\!\!*$) and weakest precondition ($\mathsf{wp}\ e\ \{\Phi\}$)
Session types	Actris dependent separation protocols (iProto)
Unique references	Points-to connective ($\ell \mapsto v$)
Shared references	Invariants ($\boxed{P}$)
Copyable types	Persistent modality ($\square\, P$)
Lock types	Iris's lock library

Overview of features

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

Linear products	Separation Conjunction ($*$)
Function types	Wand ($-\!*$) and weakest precondition ($\text{wp}\ e\ \{\Phi\}$)
Session types	Actris dependent separation protocols (iProto)
Unique references	Points-to connective ($\ell \mapsto v$)
Shared references	Invariants ($\boxed{P}$)
Copyable types	Persistent modality ($\Box\, P$)
Lock types	Iris's lock library
Session choice types	Actris dependent separation protocols (iProto)

Overview of features

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

Linear products	Separation Conjunction ($*$)
Function types	Wand ($-\!\!*$) and weakest precondition ($\text{wp}\ e\ \{\Phi\}$)
Session types	Actris dependent separation protocols (iProto)
Unique references	Points-to connective ($\ell \mapsto v$)
Shared references	Invariants ($\boxed{P}$)
Copyable types	Persistent modality ($\Box P$)
Lock types	Iris's lock library
Session choice types	Actris dependent separation protocols (iProto)
Recursion	Guarded step-indexed recursion ($\triangleright$)

Overview of features

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

Linear products	Separation Conjunction ($*$)
Function types	Wand ($-\!\!*$) and weakest precondition ($\text{wp}\ e\ \{\Phi\}$)
Session types	Actris dependent separation protocols (iProto)
Unique references	Points-to connective ($\ell \mapsto v$)
Shared references	Invariants ($\boxed{P}$)
Copyable types	Persistent modality ($\Box P$)
Lock types	Iris's lock library
Session choice types	Actris dependent separation protocols (iProto)
Recursion	Guarded step-indexed recursion ($\triangleright$)
Term polymorphism	Higher-order impredicative quantifiers

Overview of features

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

Linear products	Separation Conjunction ($*$)
Function types	Wand ($-\!\!*$) and weakest precondition ($\mathrm{wp}\ e\ \{\Phi\}$)
Session types	Actris dependent separation protocols (iProto)
Unique references	Points-to connective ($\ell \mapsto v$)
Shared references	Invariants ($\boxed{P}$)
Copyable types	Persistent modality ($\square P$)
Lock types	Iris's lock library
Session choice types	Actris dependent separation protocols (iProto)
Recursion	Guarded step-indexed recursion ($\triangleright$)
Term polymorphism	Higher-order impredicative quantifiers
Session polymorphism	Higher-order impredicative protocols binders

Overview of features

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

Linear products	Separation Conjunction ($*$)
Function types	Wand ($-\!*$) and weakest precondition ($\text{wp}\ e\ \{\Phi\}$)
Session types	Actris dependent separation protocols (iProto)
Unique references	Points-to connective ($\ell \mapsto v$)
Shared references	Invariants ($\boxed{P}$)
Copyable types	Persistent modality ($\Box P$)
Lock types	Iris's lock library
Session choice types	Actris dependent separation protocols (iProto)
Recursion	Guarded step-indexed recursion ($\triangleright$)
Term polymorphism	Higher-order impredicative quantifiers
Session polymorphism	Higher-order impredicative protocols binders
Term subtyping	Predicates closed under wand ($\forall v.\ A_1\ v -\!*\ A_2\ v$)

Overview of features

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

Linear products	Separation Conjunction ($*$)
Function types	Wand ($-\!*$) and weakest precondition ($\text{wp } e \{\Phi\}$)
Session types	Actris dependent separation protocols (iProto)
Unique references	Points-to connective ($\ell \mapsto v$)
Shared references	Invariants ($\boxed{P}$)
Copyable types	Persistent modality ($\Box P$)
Lock types	Iris's lock library
Session choice types	Actris dependent separation protocols (iProto)
Recursion	Guarded step-indexed recursion ($\triangleright$)
Term polymorphism	Higher-order impredicative quantifiers
Session polymorphism	Higher-order impredicative protocols binders
Term subtyping	Predicates closed under wand ($\forall v.\, A_1\ v -\!*\ A_2\ v$)
Session subtyping	Actris 2.0 subprotocols ($\sqsubseteq$)

Overview of features - Definitions

Shared references: $\texttt{ref}_{\texttt{shr}}\; A \triangleq \lambda\, w.\, (w \in \mathsf{Loc}) * \boxed{\exists v.\, (w \mapsto v) * \Box(A\, v)}$

Copyable types: $\texttt{copy}\, A \triangleq \lambda\, w.\; \Box(A\, w)$

Lock types:

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

Session choice:

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

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

Polymorphism:

$$\forall (X : k).\, A \triangleq \lambda\, w.\, \forall (X : \mathsf{Type}_k).\, \mathsf{wp}\; w\; ()\; \{A\}$$
$$\exists (X : k).\, A \triangleq \lambda\, w.\, \exists (X : \mathsf{Type}_k).\, \triangleright(A\, w)$$
$$\mathbf{!}_{\vec{X}:\vec{k}}\, A.\, S \triangleq \mathbf{!}\, (\vec{X} : \vec{\mathsf{Type}}_k)(v : \mathsf{Val})\, \langle v \rangle \{A\, v\}.\, S$$
$$\mathbf{?}_{\vec{X}:\vec{k}}\, A.\, S \triangleq \mathbf{?}\, (\vec{X} : \vec{\mathsf{Type}}_k)(v : \mathsf{Val})\, \langle v \rangle \{A\, v\}.\, S$$

Term subtyping: $A <: B \triangleq \forall v.\, A\; v \twoheadrightarrow B\; v$

Session subtyping: $S_1 <: S_2 \triangleq S_1 \sqsubseteq S_2$

Typing the Untypeable

An Untypeable Program

Consider the following judgement:

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

An Untypeable Program

Consider the following judgement:

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

Is it typeable?

An Untypeable Program

Consider the following judgement:

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

Is it typeable? No

An Untypeable Program

Consider the following judgement:

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

Is it typeable? No It violates the ownership discipline

An Untypeable Program

Consider the following judgement:

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

Is it typeable? No It violates the ownership discipline

Is it safe?

An Untypeable Program

Consider the following judgement:

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

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

An Untypeable Program

Consider the following judgement:

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

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 judgement:

$$\vDash \lambda c. (\texttt{recv}\ c \parallel \texttt{recv}\ c) : \texttt{chan}\ (\textbf{?}\mathbb{Z}. \textbf{?}\mathbb{Z}. \texttt{end}) \multimap (\mathbb{Z} \times \mathbb{Z})$$

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

An Untypeable Program

Consider the following judgement:

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

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 judgement:

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

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

An Untypeable Program

Consider the following judgement:

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

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 judgement:

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

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

$$(c \rightarrowtail \mathbf{?}\,(v_1 : \mathsf{Val})\,\langle v_1\rangle\{v_1 \in \mathbb{Z}\}.\, \mathbf{?}\,(v_2 : \mathsf{Val})\,\langle v_2\rangle\{v_2 \in \mathbb{Z}\}.\, \mathsf{end}) \twoheadrightarrow$$
$$\mathsf{wp}\ (\texttt{recv}\ c \parallel \texttt{recv}\ c)\ \{v.\, \exists v_1, v_2.\, (v = (v_1, v_2)) * \triangleright(v_1 \in \mathbb{Z}) * \triangleright(v_2 \in \mathbb{Z})\}$$

An Untypeable Program

Consider the following judgement:

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

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

$$(c \rightarrowtail \textbf{?}\,(v_1 : \mathsf{Val})\,\langle v_1\rangle\{v_1 \in \mathbb{Z}\}.\,\textbf{?}\,(v_2 : \mathsf{Val})\,\langle v_2\rangle\{v_2 \in \mathbb{Z}\}.\,\mathsf{end}) \twoheadrightarrow$$
$$\mathsf{wp}\,(\texttt{recv}\ c \parallel \texttt{recv}\ c)\,\{v.\, \exists v_1, v_2.\, (v = (v_1, v_2)) * \triangleright(v_1 \in \mathbb{Z}) * \triangleright(v_2 \in \mathbb{Z})\}$$

And then using Iris's ghost state machinery!

An Untypeable Program

Consider the following judgement:

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

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 proven by unfolding all type-level definitions

$$(c \rightarrowtail \textbf{?}\,(v_1 : \mathsf{Val})\,\langle v_1\rangle\{v_1 \in \mathbb{Z}\}.\,\textbf{?}\,(v_2 : \mathsf{Val})\,\langle v_2\rangle\{v_2 \in \mathbb{Z}\}.\,\mathsf{end}) \mathrel{-\!\!*}$$
$$\mathsf{wp}\ (\texttt{recv}\ c \mid\mid \texttt{recv}\ c)\ \{v.\, \exists v_1, v_2.\, (v = (v_1, v_2)) * \triangleright(v_1 \in \mathbb{Z}) * \triangleright(v_2 \in \mathbb{Z})\}$$

And then using Iris's ghost state machinery! 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

Material:

- Paper on semantic session type system (TBD)
- Mechanisation in Coq (https://gitlab.mpi-sws.org/iris/actris/-/tree/cpp21)

Questions?

Asynchronous Session Subtyping

Semantic Asynchronous Session Subtyping

Conventional session subtyping:

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

Semantic Asynchronous Session Subtyping

Conventional session subtyping:

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

Asynchronous session subtyping:

$$\mathbf{?}A_1.\,\mathbf{!}A_2.\,S <: \mathbf{!}A_2.\,\mathbf{?}A_1.\,S$$

Semantic Asynchronous Session Subtyping

Conventional session subtyping:

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

Asynchronous session subtyping:

$$\mathbf{?}A_1.\, \mathbf{!}A_2.\, S <: \mathbf{!}A_2.\, \mathbf{?}A_1.\, S$$

Polymorphic session subtyping:

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

Semantic Asynchronous Session Subtyping - Example

Goal:

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

Semantic Asynchronous Session Subtyping - Example

Goal:

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

Derivation:

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

Semantic Asynchronous Session Subtyping - Example

Goal:

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

Derivation:

$$\begin{aligned} &\mu\,(\mathit{rec} : \blacklozenge).\,\mathbf{!}_{(X,Y:\bigstar)}\,(X \multimap Y).\,\mathbf{!}X.\,\mathbf{?}Y.\,\mathit{rec} \\ <:\ &\mu\,(\mathit{rec} : \blacklozenge).\,\mathbf{!}_{(X_1,Y_1:\bigstar)}\,(X_1 \multimap Y_1).\,\mathbf{!}X_1.\,\mathbf{?}Y_1.\,\mathbf{!}_{(X_2,Y_2:\bigstar)}\,(X_2 \multimap Y_2).\,\mathbf{!}X_2.\,\mathbf{?}Y_2.\,\mathit{rec} && (\text{LÖB}) \end{aligned}$$

Semantic Asynchronous Session Subtyping - Example

Goal:

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

Derivation:

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

$$<: \mu\,(\mathit{rec} : \blacklozenge).\,\mathbf{!}_{(X_1,Y_1:\bigstar)}\,(X_1 \multimap Y_1).\,\mathbf{!}X_1.\,\mathbf{?}Y_1.\,\mathbf{!}_{(X_2,Y_2:\bigstar)}\,(X_2 \multimap Y_2).\,\mathbf{!}X_2.\,\mathbf{?}Y_2.\,\mathit{rec} \qquad (\text{LÖB})$$

$$<: \mu\,(\mathit{rec} : \blacklozenge).\,\mathbf{!}_{(X_1,X_2:\bigstar)}\,(X_1 \multimap \mathsf{B}).\,\mathbf{!}X_1.\,\mathbf{?}\mathsf{B}.\,\mathbf{!}(X_2 \multimap \mathsf{Z}).\,\mathbf{!}X_2.\,\mathbf{?}\mathsf{Z}.\,\mathit{rec} \qquad (\text{S-ELIM}, \text{S-INTRO})$$

Rules:

S-ELIM

$$\frac{S_1 <: \mathbf{!}A.\,S_2}{S_1 <: \mathbf{!}_{(\vec{X}:\vec{k})}A.\,S_2}$$

S-INTRO

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

Semantic Asynchronous Session Subtyping - Example

Goal:

$$\mu\,(rec : \blacklozenge).\,!_{(X,Y:\bigstar)}\,(X \multimap Y).\,!X.\,?Y.\,rec <: \mu\,(rec : \blacklozenge).\,!_{(X_1,X_2:\bigstar)}\,(X_1 \multimap \mathsf{B}).\,!X_1.\,!(X_2 \multimap \mathsf{Z}).\,!X_2.\,?\mathsf{B}.\,?\mathsf{Z}.\,rec$$

Derivation:

$$\begin{aligned}
&\mu\,(rec : \blacklozenge).\,!_{(X,Y:\bigstar)}\,(X \multimap Y).\,!X.\,?Y.\,rec \\
<:\;&\mu\,(rec : \blacklozenge).\,!_{(X_1,Y_1:\bigstar)}\,(X_1 \multimap Y_1).\,!X_1.\,?Y_1.\,!_{(X_2,Y_2:\bigstar)}\,(X_2 \multimap Y_2).\,!X_2.\,?Y_2.\,rec && (\text{LÖB}) \\
<:\;&\mu\,(rec : \blacklozenge).\,!_{(X_1,X_2:\bigstar)}\,(X_1 \multimap \mathsf{B}).\,!X_1.\,?\mathsf{B}.\,!(X_2 \multimap \mathsf{Z}).\,!X_2.\,?\mathsf{Z}.\,rec && (\text{S-ELIM}, \text{S-INTRO}) \\
<:\;&\mu\,(rec : \blacklozenge).\,!_{(X_1,X_2:\bigstar)}\,(X_1 \multimap \mathsf{B}).\,!X_1.\,!(X_2 \multimap \mathsf{Z}).\,?\mathsf{B}.\,!X_2.\,?\mathsf{Z}.\,rec && (\text{SWAP})
\end{aligned}$$

Rules:

S-ELIM

$$\frac{S_1 <: !A.\,S_2}{S_1 <: !_{(\vec{X}:\vec{k})}A.\,S_2}$$

S-INTRO

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

SWAP

$$?A_1.\,!A_2.\,S <: !A_2.\,?A_1.\,S$$

Semantic Asynchronous Session Subtyping - Example

Goal:

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

Derivation:

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

$$<: \mu\,(\mathit{rec} : \blacklozenge).\, \mathbf{!}_{(X_1,Y_1:\bigstar)}\,(X_1 \multimap Y_1).\, \mathbf{!}X_1.\, \mathbf{?}Y_1.\, \mathbf{!}_{(X_2,Y_2:\bigstar)}\,(X_2 \multimap Y_2).\, \mathbf{!}X_2.\, \mathbf{?}Y_2.\, \mathit{rec} \qquad (\text{LÖB})$$

$$<: \mu\,(\mathit{rec} : \blacklozenge).\, \mathbf{!}_{(X_1,X_2:\bigstar)}\,(X_1 \multimap \mathsf{B}).\, \mathbf{!}X_1.\, \mathbf{?}\mathsf{B}.\, \mathbf{!}(X_2 \multimap \mathsf{Z}).\, \mathbf{!}X_2.\, \mathbf{?}\mathsf{Z}.\, \mathit{rec} \qquad (\text{S-ELIM}, \text{S-INTRO})$$

$$<: \mu\,(\mathit{rec} : \blacklozenge).\, \mathbf{!}_{(X_1,X_2:\bigstar)}\,(X_1 \multimap \mathsf{B}).\, \mathbf{!}X_1.\, \mathbf{!}(X_2 \multimap \mathsf{Z}).\, \mathbf{?}\mathsf{B}.\, \mathbf{!}X_2.\, \mathbf{?}\mathsf{Z}.\, \mathit{rec} \qquad (\text{SWAP})$$

$$<: \mu\,(\mathit{rec} : \blacklozenge).\, \mathbf{!}_{(X_1,X_2:\bigstar)}\,(X_1 \multimap \mathsf{B}).\, \mathbf{!}X_1.\, \mathbf{!}(X_2 \multimap \mathsf{Z}).\, \mathbf{!}X_2.\, \mathbf{?}\mathsf{B}.\, \mathbf{?}\mathsf{Z}.\, \mathit{rec} \qquad (\text{SWAP})$$

Rules:

S-ELIM

$$\frac{S_1 <: \mathbf{!}A.\, S_2}{S_1 <: \mathbf{!}_{(\vec{X}:\vec{k})} A.\, S_2}$$

S-INTRO

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

SWAP

$$\mathbf{?}A_1.\, \mathbf{!}A_2.\, S <: \mathbf{!}A_2.\, \mathbf{?}A_1.\, S$$