Machine-Checked Semantic Session Typing

Jonas Kastberg Hinrichsen, Aarhus University

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

November 22, 2021
Aarhus University, Denmark

Agenda

- Session Types
- Semantic Typing (vs Syntactic Typing)
- Semantic Session Type System

(Shared memory) concurrency is hard

Race conditions on shared resources (e.g. references)

- Potentially many program interleavings

(Shared memory) concurrency is hard

Race conditions on shared resources (e.g. references)

- Potentially many program interleavings

Non-trivial memory models

- Strong memory - Less optimisations
- Weak memory - Unintuitive (/undefined) behaviour

(Shared memory) concurrency is hard

Race conditions on shared resources (e.g. references)

- Potentially many program interleavings

Non-trivial memory models

- Strong memory - Less optimisations
- Weak memory - Unintuitive (/undefined) behaviour

Typing disciplines

(Shared memory) concurrency is hard

Race conditions on shared resources (e.g. references)

- Potentially many program interleavings

Non-trivial memory models

- Strong memory - Less optimisations
- Weak memory - Unintuitive (/undefined) behaviour

Typing disciplines come with restrictions

- Linear types - resources can only occur in one thread

(Shared memory) concurrency is hard

Race conditions on shared resources (e.g. references)

- Potentially many program interleavings

Non-trivial memory models

- Strong memory - Less optimisations
- Weak memory - Unintuitive (/undefined) behaviour

Typing disciplines come with restrictions

- Linear types - resources can only occur in one thread
  - No concurrency

The message passing model

Principled way of writing concurrent programs

The message passing model

Principled way of writing concurrent programs

- Isolation of concurrent behaviour

The message passing model

Principled way of writing concurrent programs

- Isolation of concurrent behaviour
- Threads as services and clients

The message passing model

Principled way of writing concurrent programs

- Isolation of concurrent behaviour
- Threads as services and clients
- Used in Erlang, Elixir, Go, Java, Scala, F#, C#, and more

The message passing model

Principled way of writing concurrent programs

- Isolation of concurrent behaviour
- Threads as services and clients
- Used in Erlang, Elixir, Go, Java, Scala, F#, C#, and more

Message passing primitives

`new_chan` (), `send` c v, `recv` c

The message passing model

Principled way of writing concurrent programs

- Isolation of concurrent behaviour
- Threads as services and clients
- Used in Erlang, Elixir, Go, Java, Scala, F#, C#, and more

Message passing primitives

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

Example:

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

The message passing model

Principled way of writing concurrent programs

- Isolation of concurrent behaviour
- Threads as services and clients
- Used in Erlang, Elixir, Go, Java, Scala, F#, C#, and more

Message passing primitives

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

Example:

$$
\begin{array}{l}
\texttt{let}\,(c, c') := \texttt{new\_chan}\ ()\ \texttt{in} \\
\texttt{fork}\ \left\{\texttt{let}\, x := \texttt{recv}\ c'\ \texttt{in}\,\texttt{send}\ c'\ (x+2)\right\}; \\
\texttt{send}\ c\ 40;\ \texttt{recv}\ c
\end{array}
$$

Many variants of message passing exist

We consider: asynchronous, order-preserving and reliable

(Binary) Session Types - A type system for message passing

Syntax

$$
\begin{aligned}
S ::= \ & \mathbf{!}A.\,S & \mid \\
& \mathbf{?}A.\,S & \mid \\
& \texttt{end} & \mid \ \ldots
\end{aligned}
$$

$$A ::= \ \mathbf{Z} \mid \mathbf{1} \mid \texttt{chan}\ S \mid \ldots$$

(Binary) Session Types - A type system for message passing

Syntax

$$
\begin{aligned}
S ::= \;& \mathbf{!}A.\,S && | \\
& \mathbf{?}A.\,S && | \\
& \texttt{end} && | \ \ldots
\end{aligned}
$$

$$A ::= \;\mathbf{Z} \mid \mathbf{1} \mid \texttt{chan}\ S \mid \ldots$$

Type example

$$\texttt{chan}\ (\mathbf{?Z}.\,\mathbf{!Z}.\,\texttt{end})$$

(Binary) Session Types - A type system for message passing

Syntax

$$
\begin{aligned}
S ::= \;& \mathbf{!}A.\,S & \mid \\
& \mathbf{?}A.\,S & \mid \\
& \texttt{end} & \mid \ldots
\end{aligned}
$$

$$A ::= \mathbf{Z} \mid \mathbf{1} \mid \texttt{chan}\ S \mid \ldots$$

Type example

$$\texttt{chan}\ (\mathbf{?Z}.\,\mathbf{!Z}.\,\texttt{end})$$

Usage

$$c : \texttt{chan}\ S$$

(Binary) Session Types - A type system for message passing

Syntax

$$S ::= \;\mathbf{!}A.\,S \;\mid\; \mathbf{?}A.\,S \;\mid\; \texttt{end} \;\mid \ldots$$

$$A ::= \;\mathbf{Z} \mid \mathbf{1} \mid \texttt{chan}\ S \mid \ldots$$

Type example

$$\texttt{chan}\ (\mathbf{?Z}.\,\mathbf{!Z}.\,\texttt{end})$$

Usage

$$c : \texttt{chan}\ S$$

Duality

$$\begin{aligned}
\overline{\mathbf{!}A.\,S} &= \mathbf{?}A.\,\overline{S} \\
\overline{\mathbf{?}A.\,S} &= \mathbf{!}A.\,\overline{S} \\
\overline{\texttt{end}} &= \texttt{end}
\end{aligned}$$

(Binary) Session Types - A type system for message passing

Syntax

$$S ::= \mathbf{!}A.\,S \mid \mathbf{?}A.\,S \mid \mathtt{end} \mid \ldots$$

$$A ::= \mathbf{Z} \mid \mathbf{1} \mid \mathtt{chan}\ S \mid \ldots$$

Type example

$$\mathtt{chan}\ (\mathbf{?Z}.\,\mathbf{!Z}.\,\mathtt{end})$$

Usage

$$c : \mathtt{chan}\ S$$

Duality

$$\begin{aligned} \overline{\mathbf{!}A.\,S} &= \mathbf{?}A.\,\overline{S} \\ \overline{\mathbf{?}A.\,S} &= \mathbf{!}A.\,\overline{S} \\ \overline{\mathtt{end}} &= \mathtt{end} \end{aligned}$$

Rules

$$\Gamma \vdash \mathtt{new\_chan}\ () : \mathtt{chan}\ S \times \mathtt{chan}\ \overline{S} \dashv \Gamma$$

(Binary) Session Types - A type system for message passing

Syntax

$$S ::= \,!A.\,S \mid ?A.\,S \mid \mathtt{end} \mid \ldots$$

$$A ::= \mathbf{Z} \mid \mathbf{1} \mid \mathtt{chan}\ S \mid \ldots$$

Type example

$$\mathtt{chan}\ (?\mathbf{Z}.\,!\mathbf{Z}.\,\mathtt{end})$$

Usage

$$c : \mathtt{chan}\ S$$

Duality

$$\begin{aligned} \overline{!A.\,S} &= ?A.\,\overline{S} \\ \overline{?A.\,S} &= !A.\,\overline{S} \\ \overline{\mathtt{end}} &= \mathtt{end} \end{aligned}$$

Rules

$$\Gamma \vdash \mathtt{new\_chan}\ () : \mathtt{chan}\ S \times \mathtt{chan}\ \overline{S} \dashv \Gamma$$

$$\Gamma, x : \mathtt{chan}\ (!A.\,S), y : A \vdash \mathtt{send}\ x\ y : \mathbf{1} \dashv \Gamma, x : \mathtt{chan}\ S$$

(Binary) Session Types - A type system for message passing

Syntax

$$S ::= \mathbf{!}A.\,S \mid \mathbf{?}A.\,S \mid \mathtt{end} \mid \dots$$

$$A ::= \mathbf{Z} \mid \mathbf{1} \mid \mathtt{chan}\ S \mid \dots$$

Type example

$$\mathtt{chan}\ (\mathbf{?Z}.\,\mathbf{!Z}.\,\mathtt{end})$$

Usage

$$c : \mathtt{chan}\ S$$

Duality

$$\begin{aligned} \overline{\mathbf{!}A.\,S} &= \mathbf{?}A.\,\overline{S} \\ \overline{\mathbf{?}A.\,S} &= \mathbf{!}A.\,\overline{S} \\ \overline{\mathtt{end}} &= \mathtt{end} \end{aligned}$$

Rules

$$\Gamma \vdash \mathtt{new\_chan}\ () : \mathtt{chan}\ S \times \mathtt{chan}\ \overline{S} \dashv \Gamma$$
$$\Gamma, x : \mathtt{chan}\ (\mathbf{!}A.\,S), y : A \vdash \mathtt{send}\ x\ y : \mathbf{1} \dashv \Gamma, x : \mathtt{chan}\ S$$
$$\Gamma, x : \mathtt{chan}\ (\mathbf{?}A.\,S) \vdash \mathtt{recv}\ x : A \dashv \Gamma, x : \mathtt{chan}\ S$$

(Binary) Session Types - A type system for message passing

Syntax

$$S ::= \mathbf{!}A.\,S \mid \mathbf{?}A.\,S \mid \mathtt{end} \mid \ldots$$

$$A ::= \mathbf{Z} \mid \mathbf{1} \mid \mathtt{chan}\ S \mid \ldots$$

Type example

$$\mathtt{chan}\ (\mathbf{?Z}.\,\mathbf{!Z}.\,\mathtt{end})$$

Usage

$$c : \mathtt{chan}\ S$$

Duality

$$\begin{aligned} \overline{\mathbf{!}A.\,S} &= \mathbf{?}A.\,\overline{S} \\ \overline{\mathbf{?}A.\,S} &= \mathbf{!}A.\,\overline{S} \\ \overline{\mathtt{end}} &= \mathtt{end} \end{aligned}$$

Rules

$$\Gamma \vdash \mathtt{new\_chan}\ () : \mathtt{chan}\ S \times \mathtt{chan}\ \overline{S} \dashv \Gamma$$
$$\Gamma, x : \mathtt{chan}\ (\mathbf{!}A.\,S), y : A \vdash \mathtt{send}\ x\ y : \mathbf{1} \dashv \Gamma, x : \mathtt{chan}\ S$$
$$\Gamma, x : \mathtt{chan}\ (\mathbf{?}A.\,S) \vdash \mathtt{recv}\ x : A \dashv \Gamma, x : \mathtt{chan}\ S$$

Program example

$$\begin{aligned} &\lambda c.\, \mathtt{let}\ x := \mathtt{recv}\ c\ \mathtt{in} \\ &\quad \mathtt{send}\ c\ (x + 2) \end{aligned}$$

(Binary) Session Types - A type system for message passing

Syntax

$$
\begin{aligned}
S ::=\ & !A.\,S & | \\
& ?A.\,S & | \\
& \mathtt{end} & | \ \ldots
\end{aligned}
$$

$$A ::= \ \mathbf{Z} \mid \mathbf{1} \mid \mathtt{chan}\ S \mid \ldots$$

Type example

$$\mathtt{chan}\ (?\mathbf{Z}.\,!\mathbf{Z}.\,\mathtt{end})$$

Usage

$$c : \mathtt{chan}\ S$$

Duality

$$
\begin{aligned}
\overline{!A.\,S} &= ?A.\,\overline{S} \\
\overline{?A.\,S} &= !A.\,\overline{S} \\
\overline{\mathtt{end}} &= \mathtt{end}
\end{aligned}
$$

Rules

$$\Gamma \vdash \mathtt{new\_chan}\ () : \mathtt{chan}\ S \times \mathtt{chan}\ \overline{S} \dashv \Gamma$$

$$\Gamma, x : \mathtt{chan}\ (!A.\,S), y : A \vdash \mathtt{send}\ x\ y : \mathbf{1} \dashv \Gamma, x : \mathtt{chan}\ S$$

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

Program example

$$
\begin{aligned}
\Gamma \vdash \lambda c.\ & \mathtt{let}\ x := \mathtt{recv}\ c\ \mathtt{in} \\
& \mathtt{send}\ c\ (x + 2) : \mathtt{chan}\ (?\mathbf{Z}.\,!\mathbf{Z}.\,\mathtt{end}) \multimap \mathbf{1} \dashv \Gamma
\end{aligned}
$$

Session Types - The bigger picture

Active topic of research since 1993

Has been used to guarantee intricate properties

- Deadlock freedom
- Session fidelity
  - Programs behave according to a session type

Has been scaled to bigger problems

- Multi-Party Session Types

Has been applied to industry-level languages

- C, Haskell, Java, OCaml, Rust, Scala
- `https://groups.inf.ed.ac.uk/abcd/session-implementations.html`

Problems

1. **Lack of feature-rich session type systems**
   - Polymorphism, recursion, and subtyping have been studied individually
   - No session type systems that combines all three

Problems

1. **Lack of feature-rich session type systems**
   - Polymorphism, recursion, and subtyping have been studied individually
   - No session type systems that combines all three
2. **No support for "racy" yet safe programs**
   - Session type systems enforce a strict ownership discipline of channel endpoints
   - No way to type check safe sharing of channel endpoints

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

Problems

1. **Lack of feature-rich session type systems**
   - Polymorphism, recursion, and subtyping have been studied individually
   - No session type systems that combines all three
2. **No support for “racy” yet safe programs**
   - Session type systems enforce a strict ownership discipline of channel endpoints
   - No way to type check safe sharing of channel endpoints

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

3. **Lack of mechanised soundness proofs for session type systems**
   - Few results exist for simpler systems
   - None exist for more expressive systems

Semantic typing

Semantic typing [Milner, Ahmed, Princeton PCC project, RustBelt project]

- Type system defined in terms of language semantics
- Modernly defined in terms of a program logic
- Expressivity and soundness inherited from underlying logic
- Allows manually proving safe yet untypeable programs

Key Idea

Semantic typing using **Iris**

Semantic typing [Milner, Ahmed, Princeton PCC project, RustBelt project]

- Type system defined in terms of language semantics
- Modernly defined in terms of a program logic
- Expressivity and soundness inherited from underlying logic
- Allows manually proving safe yet untypeable programs

Iris [Iris project]

- Higher-order concurrent separation logic
- Mechanised in **Coq**, with tactic support

Semantic typing using **Iris** and **Actris**

Semantic typing [Milner, Ahmed, Princeton PCC project, RustBelt project]

- Type system defined in terms of language semantics
- Modernly defined in terms of a program logic
- Expressivity and soundness inherited from underlying logic
- Allows manually proving safe yet untypeable programs

Iris [Iris project]

- Higher-order concurrent separation logic
- Mechanised in **Coq**, with tactic support

Actris [Hinrichsen et al. POPL'20]

- **Dependent separation protocols:** Logical protocols inspired by session types
- Mechanised in **Coq**, with tactic support

Machine-Checked Semantic Session Type System

Machine-Checked Semantic Session Type System

1. 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

Machine-Checked Semantic Session Type System

1. 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
2. Supports integrating safe yet untypeable programs, through manual proofs

Machine-Checked Semantic Session Type System

1. 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
2. Supports integrating safe yet untypeable programs, through manual proofs
3. Full mechanisation in Coq (https://gitlab.mpi-sws.org/iris/actris/-/tree/cpp21)

Syntactic Typing
vs.
Semantic Typing

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: $A ::= \mathbf{Z} \mid A_1 \times A_2 \mid \ldots$

Syntactic Typing

In a **syntactic type system**

- **Types** are defined as a closed inductive definition: $A ::= \mathbf{Z} \mid A_1 \times A_2 \mid \ldots$
- **Rules** are defined as a closed inductive relation

Syntactic Typing

In a **syntactic type system**

- **Types** are defined as a closed inductive definition: $A ::= \mathbf{Z} \mid A_1 \times A_2 \mid \ldots$
- **Rules** are defined as a closed inductive relation: $\vdash i : \mathbf{Z}$

Syntactic Typing

In a **syntactic type system**

- **Types** are defined as a closed inductive definition: $A ::= \mathbf{Z} \mid A_1 \times A_2 \mid \ldots$
- **Rules** are defined as a closed inductive relation: $\vdash i : \mathbf{Z}$
- **Type safety**

Syntactic Typing

In a **syntactic type system**

- **Types** are defined as a closed inductive definition: $A ::= \mathbf{Z} \mid A_1 \times A_2 \mid \ldots$
- **Rules** are defined as a closed inductive relation: $\vdash i : \mathbf{Z}$
- **Type safety**: if $\vdash e : A$ then safe e

Syntactic Typing

In a **syntactic type system**

- **Types** are defined as a closed inductive definition: $A ::= \mathbf{Z} \mid A_1 \times A_2 \mid \ldots$
- **Rules** are defined as a closed inductive relation: $\vdash i : \mathbf{Z}$
- **Type safety**: if $\vdash e : A$ then safe e

$$\text{safe } e \triangleq \forall e'.\ \text{if } e \longrightarrow^* e' \text{ then } (e' \in \text{Val}) \text{ or } (\exists e''.\ e' \longrightarrow e'')$$

Syntactic Typing

In a **syntactic type system**

- **Types** are defined as a closed inductive definition: $A ::= \mathbf{Z} \mid A_1 \times A_2 \mid \ldots$
- **Rules** are defined as a closed inductive relation: $\vdash i : \mathbf{Z}$
- **Type safety**: if $\vdash e : A$ then safe e

  $$\text{safe } e \triangleq \forall e'.\ \text{if } e \longrightarrow^* e' \text{ then } (e' \in \text{Val}) \text{ or } (\exists e''.\, e' \longrightarrow e'')$$
- **Type safety** is proven using **progress** and **preservation**

Syntactic Typing

In a **syntactic type system**

- **Types** are defined as a closed inductive definition: $A ::= \mathbf{Z} \mid A_1 \times A_2 \mid \ldots$
- **Rules** are defined as a closed inductive relation: $\vdash i : \mathbf{Z}$
- **Type safety**: if $\vdash e : A$ then safe e

  $$\text{safe } e \triangleq \forall e'.\ \text{ if } e \longrightarrow^* e' \text{ then } (e' \in \text{Val}) \text{ or } (\exists e''.\, e' \longrightarrow e'')$$
- **Type safety** is proven using **progress** and **preservation**
  - **Progress**: if $\vdash e : A$ then $(e \in \text{Val})$ or $(\exists e'.\, e \longrightarrow e')$

Syntactic Typing

In a **syntactic type system**

- **Types** are defined as a closed inductive definition: $A ::= \mathbf{Z} \mid A_1 \times A_2 \mid \ldots$
- **Rules** are defined as a closed inductive relation: $\vdash i : \mathbf{Z}$
- **Type safety**: if $\vdash e : A$ then safe e

  $$\text{safe } e \triangleq \forall e'.\ \text{if } e \longrightarrow^* e' \text{ then } (e' \in \mathsf{Val}) \text{ or } (\exists e''.\, e' \longrightarrow e'')$$
- **Type safety** is proven using **progress** and **preservation**
  - **Progress**: if $\vdash e : A$ then $(e \in \mathsf{Val})$ or $(\exists e'.\, e \longrightarrow e')$
  - **Preservation**: if $\vdash e : A$ and $e \longrightarrow e'$ then $\vdash e' : A$

Semantic Typing

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

Semantic Typing

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

- **Types** defined as predicates over values

Semantic Typing

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

- **Types** defined as predicates over values $\mathsf{Type} \triangleq \mathsf{Val} \to \mathsf{Prop}$

Semantic Typing

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

- **Types** defined as predicates over values $\mathsf{Type} \triangleq \mathsf{Val} \to \mathsf{Prop}$

$$\mathbf{Z} \triangleq \lambda w.\, w \in \mathbb{Z}$$

Semantic Typing

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

- **Types** defined as predicates over values $\mathsf{Type} \triangleq \mathsf{Val} \to \mathsf{Prop}$

$$\begin{aligned} \mathbf{Z} &\triangleq \lambda w.\, w \in \mathbb{Z} \\ A_1 \times A_2 &\triangleq \lambda w.\, \exists w_1, w_2.\, (w = (w_1, w_2)) \wedge (A_1\, w_1) \wedge (A_2\, w_2) \end{aligned}$$

Semantic Typing

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

- **Types** defined as predicates over values $\text{Type} \triangleq \text{Val} \to \text{Prop}$

$$\begin{aligned} \mathbf{Z} &\triangleq \lambda w.\, w \in \mathbb{Z} \\ A_1 \times A_2 &\triangleq \lambda w.\, \exists w_1, w_2.\, (w = (w_1, w_2)) \wedge (A_1\, w_1) \wedge (A_2\, w_2) \end{aligned}$$

- **Judgement** *defined* as safety-capturing evaluation

Semantic Typing

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

- **Types** defined as predicates over values $\mathsf{Type} \triangleq \mathsf{Val} \to \mathsf{Prop}$

$$\mathbf{Z} \triangleq \lambda w.\, w \in \mathbb{Z}$$
$$A_1 \times A_2 \triangleq \lambda w.\, \exists w_1, w_2.\, (w = (w_1, w_2)) \wedge (A_1\; w_1) \wedge (A_2\; w_2)$$

- **Judgement** *defined* as safety-capturing evaluation

$$\vDash e : A \;\triangleq\; \mathsf{safe}\; e \quad \text{and} \quad \forall v.\; \text{if } e \longrightarrow^* v \text{ then } A\; v$$

Semantic Typing

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

- **Types** defined as predicates over values $\mathsf{Type} \triangleq \mathsf{Val} \to \mathsf{Prop}$

$$\begin{aligned} \mathbf{Z} &\triangleq \lambda w.\, w \in \mathbb{Z} \\ A_1 \times A_2 &\triangleq \lambda w.\, \exists w_1, w_2.\, (w = (w_1, w_2)) \wedge (A_1\, w_1) \wedge (A_2\, w_2) \end{aligned}$$

- **Judgement** *defined* as safety-capturing evaluation

$$\vDash e : A \;\triangleq\; \mathsf{safe}\ e \quad \mathsf{and} \quad \forall v.\ \mathsf{if}\ e \longrightarrow^{*} v\ \mathsf{then}\ A\ v$$

- **Rules** are *proven* as lemmas:

Semantic Typing

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

- **Types** defined as predicates over values $\text{Type} \triangleq \text{Val} \to \text{Prop}$

$$\begin{aligned} \mathbf{Z} &\triangleq \lambda w.\, w \in \mathbb{Z} \\ A_1 \times A_2 &\triangleq \lambda w.\, \exists w_1, w_2.\, (w = (w_1, w_2)) \wedge (A_1\, w_1) \wedge (A_2\, w_2) \end{aligned}$$

- **Judgement** *defined* as safety-capturing evaluation

$$\vDash e : A \;\triangleq\; \text{safe } e \quad \text{and} \quad \forall v.\ \text{if } e \longrightarrow^* v \text{ then } A\ v$$

- **Rules** are *proven* as lemmas: $\vDash i : \mathbf{Z}$

Semantic Typing

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

- **Types** defined as predicates over values $\mathsf{Type} \triangleq \mathsf{Val} \to \mathsf{Prop}$

$$\mathbf{Z} \triangleq \lambda w.\, w \in \mathbb{Z}$$
$$A_1 \times A_2 \triangleq \lambda w.\, \exists w_1, w_2.\, (w = (w_1, w_2)) \wedge (A_1\, w_1) \wedge (A_2\, w_2)$$

- **Judgement** *defined* as safety-capturing evaluation

$$\vDash e : A \;\triangleq\; \mathsf{safe}\ e \quad \text{and} \quad \forall v.\ \text{if } e \longrightarrow^* v \text{ then } A\ v$$

- **Rules** are *proven* as lemmas: $\vDash i : \mathbf{Z} \quad \rightsquigarrow \quad i \in \mathbb{Z}$

Semantic Typing

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

- **Types** defined as predicates over values $\mathsf{Type} \triangleq \mathsf{Val} \to \mathsf{Prop}$

$$\begin{aligned} \mathbf{Z} &\triangleq \lambda w.\, w \in \mathbb{Z} \\ A_1 \times A_2 &\triangleq \lambda w.\, \exists w_1, w_2.\, (w = (w_1, w_2)) \wedge (A_1\, w_1) \wedge (A_2\, w_2) \end{aligned}$$

- **Judgement** *defined* as safety-capturing evaluation

$$\vDash e : A \quad \triangleq \quad \mathsf{safe}\ e \quad \mathsf{and} \quad \forall v.\ \mathsf{if}\ e \longrightarrow^* v\ \mathsf{then}\ A\ v$$

- **Rules** are *proven* as lemmas: $\vDash i : \mathbf{Z} \quad \rightsquigarrow \quad i \in \mathbb{Z}$
- **Semantic type safety**

Semantic Typing

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

- **Types** defined as predicates over values $\text{Type} \triangleq \text{Val} \to \text{Prop}$

$$\begin{aligned} \mathbf{Z} &\triangleq \lambda w.\, w \in \mathbb{Z} \\ A_1 \times A_2 &\triangleq \lambda w.\, \exists w_1, w_2.\, (w = (w_1, w_2)) \wedge (A_1\, w_1) \wedge (A_2\, w_2) \end{aligned}$$

- **Judgement** *defined* as safety-capturing evaluation

$$\vDash e : A \quad \triangleq \quad \text{safe } e \quad \text{and} \quad \forall v.\ \text{if } e \longrightarrow^* v \text{ then } A\ v$$

- **Rules** are *proven* as lemmas: $\vDash i : \mathbf{Z} \quad \rightsquigarrow \quad i \in \mathbb{Z}$
- **Semantic type safety**: If $\vDash e : A$ then safe e

Semantic Typing

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

- **Types** defined as predicates over values $\mathsf{Type} \triangleq \mathsf{Val} \to \mathsf{Prop}$

$$\begin{aligned} \mathbf{Z} &\triangleq \lambda w.\, w \in \mathbb{Z} \\ A_1 \times A_2 &\triangleq \lambda w.\, \exists w_1, w_2.\, (w = (w_1, w_2)) \wedge (A_1\, w_1) \wedge (A_2\, w_2) \end{aligned}$$

- **Judgement** *defined* as safety-capturing evaluation

$$\vDash e : A \quad \triangleq \quad \mathsf{safe}\ e \quad \mathsf{and} \quad \forall v.\ \mathsf{if}\ e \longrightarrow^* v\ \mathsf{then}\ A\ v$$

- **Rules** are *proven* as lemmas: $\vDash i : \mathbf{Z} \quad \rightsquigarrow \quad i \in \mathbb{Z}$
- **Semantic type safety**: If $\vDash e : A$ then safe e
  - Consequence of the judgement definition

Semantic Typing

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

- **Types** defined as predicates over values $\mathsf{Type} \triangleq \mathsf{Val} \to \mathsf{Prop}$

$$\begin{aligned} \mathbf{Z} &\triangleq \lambda w.\, w \in \mathbb{Z} \\ A_1 \times A_2 &\triangleq \lambda w.\, \exists w_1, w_2.\, (w = (w_1, w_2)) \wedge (A_1\, w_1) \wedge (A_2\, w_2) \end{aligned}$$

- **Judgement** *defined* as safety-capturing evaluation

$$\vDash e : A \quad \triangleq \quad \mathsf{safe}\ e \quad \text{and} \quad \forall v.\ \text{if } e \longrightarrow^* v \text{ then } A\ v$$

- **Rules** are *proven* as lemmas: $\vDash i : \mathbf{Z} \quad \rightsquigarrow \quad i \in \mathbb{Z}$
- **Semantic type safety**: If $\vDash e : A$ then $\mathsf{safe}\ e$
  - Consequence of the judgement definition

Semantic Typing

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

- **Types** defined as predicates over values Type $\triangleq$ Val $\rightarrow$ iProp

$$\mathbf{Z} \triangleq \lambda w.\ w \in \mathbb{Z}$$
$$A_1 \times A_2 \triangleq \lambda w.\ \exists w_1, w_2.\ (w = (w_1, w_2)) \wedge (A_1\ w_1) \wedge (A_2\ w_2)$$

- **Judgement** *defined* as safety-capturing evaluation

$$\vDash e : A \quad \triangleq \quad \text{safe } e \quad \text{and} \quad \forall v.\ \text{if } e \longrightarrow^* v \text{ then } A\ v$$

- **Rule**
- **Sem**
  -

Replacing Coq's Prop with Iris's iProp implicitly threads the heap:
- similar to Type $\triangleq$ Val $\rightarrow$ Heap $\rightarrow$ Prop
- but also handles step-indexing and user-defined ghost state

Semantic Typing

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

- **Types** defined as predicates over values $\mathsf{Type} \triangleq \mathsf{Val} \to \mathsf{iProp}$

> Iris's *weakest precondition* $(\mathsf{wp}\ e\ \{\Phi\})$:
> - captures that $\mathsf{safe}\ e$ and $\forall v.\ e \longrightarrow^* v$ then $\Phi\ v$
> - implicitly handles the heap and ghost state

- **Judgement** *defined* as safety-capturing evaluation

$$\vDash e : A \quad \triangleq \quad \mathsf{wp}\ e\ \{A\}$$

- **Rules** are *proven* as lemmas: $\vDash i : \mathbf{Z} \quad \rightsquigarrow \quad i \in \mathbb{Z}$
- **Semantic type safety**: If $\vDash e : A$ then $\mathsf{safe}\ e$
  - Consequence of the judgement definition

Semantic Typing

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

- **Types** defined as predicates over values $\mathsf{Type} \triangleq \mathsf{Val} \to \mathsf{iProp}$

> Iris's *weakest precondition* ($\mathsf{wp}\ e\ \{\Phi\}$):
> - captures that $\boxed{\mathsf{safe}\ e}$ and $\forall v.\ e \longrightarrow^* v$ then $\Phi\ v$
> - implicitly handles the heap and ghost state

- **Judgement** *defined* as safety-capturing evaluation

$$\vDash e : A \quad \triangleq \quad \mathsf{wp}\ e\ \{A\}$$

- **Rules** are *proven* as lemmas: $\vDash i : \mathbf{Z} \quad \rightsquigarrow \quad i \in \mathbb{Z}$
- **Semantic type safety**: If $\vDash e : A$ then $\boxed{\mathsf{safe}\ e}$
  - Consequence of the judgement definition

Semantic Typing

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

- **Types** defined as predicates over values $\mathsf{Type} \triangleq \mathsf{Val} \to \mathsf{iProp}$

$$\mathbf{Z} \triangleq \lambda w.\, w \in \mathbb{Z}$$
$$A_1 \times A_2 \triangleq \lambda w.\, \exists w_1, w_2.\, (w = (w_1, w_2)) * (A_1\, w_1) * (A_2\, w_2)$$

- **Judgement** *defined* as safety-capturing evaluation

$$\vDash e : A \quad \triangleq \quad \mathsf{wp}\; e\, \{A\}$$

- **R**
- **S**

Replacing regular conjunction ($\wedge$) with Iris's *separation conjunction* ($*$) yields a substructural product type

The *separation conjunction* ($P * Q$) states that P and Q hold for disjoint parts of the heap

Semantic Typing

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

- **Types** defined as predicates over values $\mathsf{Type} \triangleq \mathsf{Val} \to \mathsf{iProp}$

$$\begin{aligned} \mathbf{Z} &\triangleq \lambda w.\, w \in \mathbb{Z} \\ A_1 \times A_2 &\triangleq \lambda w.\, \exists w_1, w_2.\, (w = (w_1, w_2)) * (A_1\, w_1) * (A_2\, w_2) \\ \mathtt{ref}_{\mathtt{uniq}}\, A &\triangleq \lambda w.\, (w \in \mathsf{Loc}) * \exists v.\, (w \mapsto v) * (A\, v) \end{aligned}$$

- **Judgement** *defined* as safety-capturing evaluation

$$\vDash e : A \quad \triangleq \quad \mathsf{wp}\, e\, \{A\}$$

- **Ru**
- **Se**
  - Consequence of the judgement definition

The *points-to connective* ($\ell \mapsto v$) asserts exclusive ownership of a location ℓ, stating that it holds the value v

Semantic Typing

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

- **Types** defined as predicates over values $\mathsf{Type} \triangleq \mathsf{Val} \to \mathsf{iProp}$

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

- **Ju**

Adding Iris's later modality ($\triangleright$) allows modeling equi-recursive types using Iris's guarded recursion operator ($\mu X.\, A$)

- **Rules** are *proven* as lemmas: $\vDash i : \mathbf{Z} \quad \rightsquigarrow \quad i \in \mathbb{Z}$
- **Semantic type safety**: If $\vDash e : A$ then safe e
  - Consequence of the judgement definition

Semantic Typing – Typing Contexts

Session type rules warrant pre- and post-contexts:

Semantic Typing – Typing Contexts

Session type rules warrant pre- and post-contexts:

$$\Gamma, x : \texttt{chan}\ (\mathbf{!}A.\, S), y : A \vdash \texttt{send}\ x\ y : \mathbf{1} \dashv \Gamma, x : \texttt{chan}\ S$$
$$\Gamma, x : \texttt{chan}\ (\mathbf{?}A.\, S) \vdash \texttt{recv}\ x : A \dashv \Gamma, x : \texttt{chan}\ S$$

Semantic Typing – Typing Contexts

Session type rules warrant pre- and post-contexts:

$$\Gamma, x : \mathtt{chan}\ (\mathbf{!}A.\,S), y : A \vdash \mathtt{send}\ x\ y : \mathbf{1} \dashv \Gamma, x : \mathtt{chan}\ S$$
$$\Gamma, x : \mathtt{chan}\ (\mathbf{?}A.\,S) \vdash \mathtt{recv}\ x : A \dashv \Gamma, x : \mathtt{chan}\ S$$

Semantic judgement with contexts:

$$\Gamma \vDash e : A \Dashv \Gamma' \triangleq ?$$

Semantic Typing – Typing Contexts

Session type rules warrant pre- and post-contexts:

$$\Gamma, x : \texttt{chan}\ (\mathbf{!}A.\, S), y : A \vdash \texttt{send}\ x\ y : \mathbf{1} \dashv \Gamma, x : \texttt{chan}\ S$$
$$\Gamma, x : \texttt{chan}\ (\mathbf{?}A.\, S) \vdash \texttt{recv}\ x : A \dashv \Gamma, x : \texttt{chan}\ S$$

Semantic judgement with contexts:

$$\boxed{\Gamma \vDash \sigma} \triangleq ?$$

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

The *closing substitution* judgement ($\Gamma \vDash \sigma$) captures separate ownership of the type predicates in context Γ for the values in closing substitution σ.

- $\Gamma \in$ List (String $\times$ Type)
- $\sigma \in$ String $\xrightarrow{\text{fin}}$ Val

Semantic Typing – Typing Contexts

Session type rules warrant pre- and post-contexts:

$$\Gamma, x : \texttt{chan}\ (\mathbf{!}A.\,S), y : A \vdash \texttt{send}\ x\ y : \mathbf{1} \dashv \Gamma, x : \texttt{chan}\ S$$
$$\Gamma, x : \texttt{chan}\ (\mathbf{?}A.\,S) \vdash \texttt{recv}\ x : A \dashv \Gamma, x : \texttt{chan}\ S$$

Semantic judgement with contexts:

$$\Gamma \vDash \sigma \triangleq \mathop{\Huge\ast}_{(x,A)\in\Gamma} .\ A(\sigma(x))$$
$$\Gamma \vDash e : A \mathrel{=\!\!\mid} \Gamma' \triangleq ?$$

The *iterated separating conjunction* ($\mathop{\Huge\ast}$) ensures that the resources of each variable are owned separately:

$$\mathop{\Huge\ast}_{y\in y_1\ldots y_n} .\ \Phi\, y \triangleq \Phi\, y_1 * \cdots * \Phi\, y_n$$

Semantic Typing – Typing Contexts

Session type rules warrant pre- and post-contexts:

$$\Gamma, x : \texttt{chan}\ (\mathbf{!}A.\, S), y : A \vdash \texttt{send}\ x\ y : \mathbf{1} \dashv \Gamma, x : \texttt{chan}\ S$$
$$\Gamma, x : \texttt{chan}\ (\mathbf{?}A.\, S) \vdash \texttt{recv}\ x : A \dashv \Gamma, x : \texttt{chan}\ S$$

Semantic judgement with contexts:

$$\Gamma \vDash \sigma \triangleq \mathop{\Huge\ast}_{(x,A)\in\Gamma} .\ A(\sigma(x))$$
$$\Gamma \vDash e : A \mathrel{=\!\!|} \Gamma' \triangleq \forall \sigma.\, (\Gamma \vDash \sigma) \mathrel{-\!\!*} \mathrm{wp}\ e[\sigma]\ \{w.\, (A\, w) * (\Gamma' \vDash \sigma)\}$$

The *separating implication* ($\mathrel{-\!\!*}$) is used similarly to implication as:

$$\frac{P * Q \vdash R}{P \vdash Q \mathrel{-\!\!*} R} \qquad \frac{P \wedge Q \vdash R}{P \vdash Q \Rightarrow R}$$

Semantic Typing – Typing Contexts

Session type rules warrant pre- and post-contexts:

$$\Gamma, x : \texttt{chan}\ (\mathbf{!}A.\, S), y : A \vdash \texttt{send}\ x\ y : \mathbf{1} \dashv \Gamma, x : \texttt{chan}\ S$$
$$\Gamma, x : \texttt{chan}\ (\mathbf{?}A.\, S) \vdash \texttt{recv}\ x : A \dashv \Gamma, x : \texttt{chan}\ S$$

Semantic judgement with contexts:

$$\Gamma \vDash \sigma \triangleq \mathop{\Huge\ast}_{(x,A)\in\Gamma} .\ A(\sigma(x))$$
$$\Gamma \vDash e : A \,\exists\, \Gamma' \triangleq \forall \sigma.\, (\Gamma \vDash \sigma) \twoheadrightarrow \mathsf{wp}\ e[\sigma]\ \{w.\, (A\, w) \ast (\Gamma' \vDash \sigma)\}$$

Inspired by the RustBelt project

Semantic Term Types – Example proof

Rule:

$$\frac{\Gamma_1 \vDash e_1 : A_1 \dashv \Gamma_1' \qquad \Gamma_2 \vDash e_2 : A_2 \dashv \Gamma_2'}{\Gamma_1 \cdot \Gamma_2 \vDash (e_1 \mid\mid e_2) : (A_1 \times A_2) \dashv \Gamma_1' \cdot \Gamma_2'}$$

Proof:

```
Lemma ltyped_par Γ1 Γ1' Γ2 Γ2' e1 e2 A1 A2 :
  (Γ1 ⊨ e1 : A1 ⫤ Γ1') -∗ (Γ2 ⊨ e2 : A2 ⫤ Γ2') -∗
  (Γ1 ++ Γ2 ⊨ (e1 ||| e2) : (A1 * A2) ⫤ Γ1' ++ Γ2').
Proof.
  iIntros "#He1 #He2 !>" (σ) "HΓ /=".
  iDestruct (ctx_ltyped_app with "HΓ")
    as "[HΓ1 HΓ2]".
  wp_apply (wp_par with "(He1 HΓ1) (He2 HΓ2)").
  iIntros (w1 w2) "[[HA1 HΓ1'] [HA2 HΓ2']] !>".
  iSplitL "HA1 HA2".
  + iExists w1, w2. by iFrame.
  + iApply ctx_ltyped_app. by iFrame.
Qed.
```

Semantic Term Types – Example proof

Rule:

$$\frac{\Gamma_1 \vDash e_1 : A_1 \dashv \Gamma_1' \qquad \Gamma_2 \vDash e_2 : A_2 \dashv \Gamma_2'}{\Gamma_1 \cdot \Gamma_2 \vDash (e_1 \mid\mid e_2) : (A_1 \times A_2) \dashv \Gamma_1' \cdot \Gamma_2'}$$

Proof:

```
Lemma ltyped_par Γ1 Γ1' Γ2 Γ2' e1 e2 A1 A2 :
  (Γ1 ⊨ e1 : A1 ⫤ Γ1') -∗ (Γ2 ⊨ e2 : A2 ⫤ Γ2') -∗
  (Γ1 ++ Γ2 ⊨ (e1 ||| e2) : (A1 * A2) ⫤ Γ1' ++ Γ2').
Proof.
  iIntros
  iDestruc
    as "[H
  wp_apply
  iIntros
  iSplitL
  + iExists w1, w2. by iFrame.
  + iApply ctx_ltyped_app. by iFrame.
Qed.
```

$$\begin{aligned}
&(\forall \sigma_1.\, (\Gamma_1 \vDash \sigma_1) \twoheadrightarrow \mathsf{wp}\ e_1[\sigma_1]\ \{w_1.\, (A_1\ w_1) * (\Gamma_1' \vDash \sigma_1)\}) \twoheadrightarrow \\
&(\forall \sigma_2.\, (\Gamma_2 \vDash \sigma_2) \twoheadrightarrow \mathsf{wp}\ e_2[\sigma_2]\ \{w_2.\, (A_2\ w_2) * (\Gamma_2' \vDash \sigma_2)\}) \twoheadrightarrow \\
&\forall \sigma.\, (\Gamma_1 \cdot \Gamma_2 \vDash \sigma) \twoheadrightarrow \mathsf{wp}\ (e_1 \mid\mid e_2)[\sigma]\ \{w.\, (\exists w_1, w_2.\, w = (w_1, w_2) * \\
&\qquad (A_1\ w_1) * (A_2\ w_2)) * \\
&\qquad (\Gamma_1' \cdot \Gamma_2' \vDash \sigma)\}
\end{aligned}$$

Semantic Term Types – Example proof

Rule:

$$\frac{\Gamma_1 \vDash e_1 : A_1 \dashv \Gamma_1' \qquad \Gamma_2 \vDash e_2 : A_2 \dashv \Gamma_2'}{\Gamma_1 \cdot \Gamma_2 \vDash (e_1 \mid\mid e_2) : (A_1 \times A_2) \dashv \Gamma_1' \cdot \Gamma_2'}$$

Proof:

```
Lemma ltyped_par Γ1 Γ1' Γ2 Γ2' e1 e2 A1 A2 :
  (Γ1 ⊨ e1 : A1 ⫤ Γ1') -∗ (Γ2 ⊨ e2 : A2 ⫤ Γ2') -∗
  (Γ1 ++ Γ2 ⊨ (e1 ||| e2) : (A1 * A2) ⫤ Γ1' ++ Γ2').
Proof.
  iIntros "#He1 #He2 !>" (σ) "HΓ /=".
  iDestruct (ctx_ltyped_app with "HΓ")
    as "[HΓ1 HΓ2]".
  wp_apply (wp_par with "(He1 HΓ1) (He2 HΓ2)").
  iIntros (w1 w2) "[[HA1 HΓ1'] [HA2 HΓ2']] !>".
  iSplitL "HA1 HA2".
  + iExists w1, w2. by iFrame.
  + iApply ctx_ltyped_app. by iFrame.
Qed.
```

```
Γ1, Γ1', Γ2, Γ2' : ctx Σ
e1, e2 : expr
A1, A2 : ltty Σ
σ : gmap string val
==========================
"He1" : Γ1 ⊨ e1 : A1 ⫤ Γ1'
"He2" : Γ2 ⊨ e2 : A2 ⫤ Γ2'
--------------------------□
"HΓ" : Γ1 ++ Γ2 ⊨ σ
--------------------------∗
WP e1[σ] ||| e2[σ]
  {{ w, (A1 × A2) w ∗
        (Γ1' ++ Γ2' ⊨ σ) }}
```

Semantic Term Types – Example proof

Rule:

$$\frac{\Gamma_1 \vDash e_1 : A_1 \dashv \Gamma_1' \qquad \Gamma_2 \vDash e_2 : A_2 \dashv \Gamma_2'}{\Gamma_1 \cdot \Gamma_2 \vDash (e_1 \mid\mid e_2) : (A_1 \times A_2) \dashv \Gamma_1' \cdot \Gamma_2'}$$

Proof:

```
Lemma ltyped_par Γ1 Γ1' Γ2 Γ2' e1 e2 A1 A2 :
  (Γ1 ⊨ e1 : A1 ⫤ Γ1') -∗ (Γ2 ⊨ e2 : A2 ⫤ Γ2') -∗
  (Γ1 ++ Γ2 ⊨ (e1 ||| e2) : (A1 * A2) ⫤ Γ1' ++ Γ2').
Proof.
  iIntros "#He1 #He2 !>" (σ) "HΓ /=".
  iDestruct (ctx_ltyped_app with "HΓ")
    as "[HΓ1 HΓ2]".
  wp_apply (wp_par with "(He1 HΓ1) (He2 HΓ2)").
  iIntros (w1 w2) "[[HA1 HΓ1'] [H
  iSplitL "HA1 HA2".
  + iExists w1, w2. by iFrame.
  + iApply ctx_ltyped_app. by iFrame.
Qed.
```

$$\frac{\Gamma_1 \cdot \Gamma_2 \vDash \sigma}{(\Gamma_1 \vDash \sigma) * (\Gamma_2 \vDash \sigma)}$$

```
Γ1, Γ1', Γ2, Γ2' : ctx Σ
e1, e2 : expr
A1, A2 : ltty Σ
σ : gmap string val
==========================
"He1" : Γ1 ⊨ e1 : A1 ⫤ Γ1'
"He2" : Γ2 ⊨ e2 : A2 ⫤ Γ2'
--------------------------□
"HΓ" : Γ1 ++ Γ2 ⊨ σ
--------------------------∗
WP e1[σ] ||| e2[σ]
  {{ w, (A1 × A2) w ∗
        (Γ1' ++ Γ2' ⊨ σ) }}
```

Semantic Term Types – Example proof

Rule:

$$\frac{\Gamma_1 \vDash e_1 : A_1 \dashv \Gamma_1' \qquad \Gamma_2 \vDash e_2 : A_2 \dashv \Gamma_2'}{\Gamma_1 \cdot \Gamma_2 \vDash (e_1 \mid\mid e_2) : (A_1 \times A_2) \dashv \Gamma_1' \cdot \Gamma_2'}$$

Proof:

```
Lemma ltyped_par Γ1 Γ1' Γ2 Γ2' e1 e2 A1 A2 :
  (Γ1 ⊨ e1 : A1 ⫤ Γ1') -∗ (Γ2 ⊨ e2 : A2 ⫤ Γ2') -∗
  (Γ1 ++ Γ2 ⊨ (e1 ||| e2) : (A1 * A2) ⫤ Γ1' ++ Γ2').
Proof.
  iIntros "#He1 #He2 !>" (σ) "HΓ /=".
  iDestruct (ctx_ltyped_app with "HΓ")
    as "[HΓ1 HΓ2]".
  wp_apply (wp_par with "(He1 HΓ1) (He2 HΓ2)").
  iIntros (w1 w2) "[[HA1 HΓ1'] [HA2 HΓ2']] !>".
  iSplitL "HA1 HA2".
  + iExists w1, w2. by iFrame.
  + iApply ctx_ltyped_app. by iFrame.
Qed.
```

```
Γ1, Γ1', Γ2, Γ2' : ctx Σ
e1, e2 : expr
A1, A2 : ltty Σ
σ : gmap string val
==========================
"He1" : Γ1 ⊨ e1 : A1 ⫤ Γ1'
"He2" : Γ2 ⊨ e2 : A2 ⫤ Γ2'
--------------------------□
"HΓ1" : Γ1 ⊨ σ
"HΓ2" : Γ2 ⊨ σ
--------------------------∗
WP e1[σ] ||| e2[σ]
  {{ w, (A1 × A2) w ∗
        (Γ1' ++ Γ2' ⊨ σ) }}
```

Semantic Term Types – Example proof

Rule:

$$\frac{\Gamma_1 \vDash e_1 : A_1 \dashv \Gamma_1' \qquad \Gamma_2 \vDash e_2 : A_2 \dashv \Gamma_2'}{\Gamma_1 \cdot \Gamma_2 \vDash (e_1 \mid\mid e_2) : (A_1 \times A_2) \dashv \Gamma_1' \cdot \Gamma_2'}$$

$$(\forall \sigma_1.\, (\Gamma_1 \vDash \sigma_1) \twoheadrightarrow \mathsf{wp}\ e_1[\sigma_1]\ \{w_1.\, (A_1\ w_1) * (\Gamma_1' \vDash \sigma_1)\})$$

$$(\forall \sigma_2.\, (\Gamma_2 \vDash \sigma_2) \twoheadrightarrow \mathsf{wp}\ e_2[\sigma_2]\ \{w_2.\, (A_2\ w_2) * (\Gamma_2' \vDash \sigma_2)\})$$

```
Proof.
  iIntros "#He1 #He2 !>" (σ) "HΓ /=".
  iDestruct (ctx_ltyped_app with "HΓ")
    as "[HΓ1 HΓ2]".
  wp_apply (wp_par with "(He1 HΓ1) (He2 HΓ2)").
  iIntros (w1 w2) "[[HA1 HΓ1'] [HA2 HΓ2']] !>".
  iSplitL "HA1 HA2".
  + iExists w1, w2. by iFrame.
  + iApply ctx_ltyped_app. by iFra
Qed.
```

```
Γ1, Γ1', Γ2, Γ2' : ctx Σ
e1, e2 : expr
A1, A2 : ltty Σ
σ : gmap string val
============================
"He1" : Γ1 ⊨ e1 : A1 ⫤ Γ1'
"He2" : Γ2 ⊨ e2 : A2 ⫤ Γ2'
----------------------------□
"HΓ1" : Γ1 ⊨ σ
"HΓ2" : Γ2 ⊨ σ
----------------------------∗
WP e1[σ] ||| e2[σ]
  {{ w, (A1 × A2) w ∗
        (Γ1' ++ Γ2' ⊨ σ) }}
```

$$\mathsf{wp}\ e_1\ \{\Phi_1\} * \mathsf{wp}\ e_2\ \{\Phi_2\} \twoheadrightarrow$$
$$\mathsf{wp}\ (e_1 \mid\mid e_2)\ \{v.\, \exists v_1, v_2.\, (v = (v_1, v_2)) * \Phi_1\ v_1 * \Phi_2\ v_2\}$$

Semantic Term Types – Example proof

Rule:

$$\frac{\Gamma_1 \vDash e_1 : A_1 \dashv \Gamma_1' \qquad \Gamma_2 \vDash e_2 : A_2 \dashv \Gamma_2'}{\Gamma_1 \cdot \Gamma_2 \vDash (e_1 \mid\mid e_2) : (A_1 \times A_2) \dashv \Gamma_1' \cdot \Gamma_2'}$$

Proof:

```
Lemma ltyped_par Γ1 Γ1' Γ2 Γ2' e1 e2 A1 A2 :
  (Γ1 ⊨ e1 : A1 ⫤ Γ1') -∗ (Γ2 ⊨ e2 : A2 ⫤ Γ2') -∗
  (Γ1 ++ Γ2 ⊨ (e1 ||| e2) : (A1 * A2) ⫤ Γ1' ++ Γ2').
Proof.
  iIntros "#He1 #He2 !>" (σ) "HΓ /=".
  iDestruct (ctx_ltyped_app with "HΓ")
    as "[HΓ1 HΓ2]".
  wp_apply (wp_par with "(He1 HΓ1) (He2 HΓ2)").
  iIntros (w1 w2) "[[HA1 HΓ1'] [HA2 HΓ2']] !>".
  iSplitL "HA1 HA2".
  + iExists w1, w2. by iFrame.
  + iApply ctx_ltyped_app. by iFrame.
Qed.
```

```
Γ1, Γ1', Γ2, Γ2' : ctx Σ
e1, e2 : expr
A1, A2 : ltty Σ
σ : gmap string val
w1, w2 : val
==========================
"He1" : Γ1 ⊨ e1 : A1 ⫤ Γ1'
"He2" : Γ2 ⊨ e2 : A2 ⫤ Γ2'
--------------------------□
"HA1" : A1 w1
"HΓ1'" : Γ1' ⊨ σ
"HA2" : A2 w2
"HΓ2'" : Γ2' ⊨ σ
--------------------------∗
(A1 × A2) (w1, w2) ∗
  (Γ1' ++ Γ2' ⊨ σ)
```

Semantic Term Types – Example proof

Rule:

$$\frac{\Gamma_1 \vDash e_1 : A_1 \dashv \Gamma_1' \qquad \Gamma_2 \vDash e_2 : A_2 \dashv \Gamma_2'}{\Gamma_1 \cdot \Gamma_2 \vDash (e_1 \mid\mid e_2) : (A_1 \times A_2) \dashv \Gamma_1' \cdot \Gamma_2'}$$

Proof:

```
Lemma ltyped_par Γ1 Γ1' Γ2 Γ2' e1 e2 A1 A2 :
  (Γ1 ⊨ e1 : A1 ⫣ Γ1') -∗ (Γ2 ⊨ e2 : A2 ⫣ Γ2') -∗
  (Γ1 ++ Γ2 ⊨ (e1 ||| e2) : (A1 * A2) ⫣ Γ1' ++ Γ2').
Proof.
  iIntros "#He1 #He2 !>" (σ) "HΓ /=".
  iDestruct (ctx_ltyped_app with "HΓ")
    as "[HΓ1 HΓ2]".
  wp_apply (wp_par with "(He1 HΓ1) (He2 HΓ2)").
  iIntros (w1 w2) "[[HA1 HΓ1'] [HA2 HΓ2']] !>".
  iSplitL "HA1 HA2".
  + iExists w1, w2. by iFrame.
  + iApply ctx_ltyped_app. by iF
Qed.
```

$$\frac{P_1 \vdash Q_1 \qquad P_2 \vdash Q_2}{P_1 * P_2 \vdash Q_1 * Q_2}$$

```
Γ1, Γ1', Γ2, Γ2' : ctx Σ
e1, e2 : expr
A1, A2 : ltty Σ
σ : gmap string val
w1, w2 : val
==========================
"He1" : Γ1 ⊨ e1 : A1 ⫣ Γ1'
"He2" : Γ2 ⊨ e2 : A2 ⫣ Γ2'
--------------------------□
"HA1" : A1 w1
"HΓ1'" : Γ1' ⊨ σ
"HA2" : A2 w2
"HΓ2'" : Γ2' ⊨ σ
--------------------------∗
(A1 × A2) (w1, w2) ∗
  (Γ1' ++ Γ2' ⊨ σ)
```

Semantic Term Types – Example proof

Rule:

$$\frac{\Gamma_1 \vDash e_1 : A_1 \dashv \Gamma_1' \qquad \Gamma_2 \vDash e_2 : A_2 \dashv \Gamma_2'}{\Gamma_1 \cdot \Gamma_2 \vDash (e_1 \mid\mid e_2) : (A_1 \times A_2) \dashv \Gamma_1' \cdot \Gamma_2'}$$

Proof:

```
Lemma ltyped_par Γ1 Γ1' Γ2 Γ2' e1 e2 A1 A2 :
  (Γ1 ⊨ e1 : A1 ⫤ Γ1') -∗ (Γ2 ⊨ e2 : A2 ⫤ Γ2') -∗
  (Γ1 ++ Γ2 ⊨ (e1 ||| e2) : (A1 * A2) ⫤ Γ1' ++ Γ2').
Proof.
  iIntros "#He1 #He2 !>" (σ) "HΓ /=".
  iDestruct (ctx_ltyped_app with "HΓ")
    as "[HΓ1 HΓ2]".
  wp_apply (wp_par with "(He1 HΓ1) (He2 HΓ2)").
  iIntros (w1 w2) "[[HA1 HΓ1'] [HA2 HΓ2']] !>".
  iSplitL "HA1 HA2".
  + iExists w1, w2. by iFrame.
  + iApply ctx_ltyped_app. by iFrame.
Qed.
```

```
Γ1, Γ1', Γ2, Γ2' : ctx Σ
e1, e2 : expr
A1, A2 : ltty Σ
σ : gmap string val
w1, w2 : val
==========================
"He1" : Γ1 ⊨ e1 : A1 ⫤ Γ1'
"He2" : Γ2 ⊨ e2 : A2 ⫤ Γ2'
--------------------------□
"HA1" : A1 w1
"HA2" : A2 w2
--------------------------∗
(A1 × A2) (w1, w2)
```

Semantic Term Types – Example proof

Rule:

$$\frac{\Gamma_1 \vDash e_1 : A_1 \dashv \Gamma_1' \qquad \Gamma_2 \vDash e_2 : A_2 \dashv \Gamma_2'}{\Gamma_1 \cdot \Gamma_2 \vDash (e_1 \mid\mid e_2) : (A_1 \times A_2) \dashv \Gamma_1' \cdot \Gamma_2'}$$

Proof:

```
Lemma ltyped_par Γ1 Γ1' Γ2 Γ2' e1 e2 A1 A2 :
  (Γ1 ⊨ e1 : A1 ⫤ Γ1') -∗ (Γ2 ⊨ e2 : A2 ⫤ Γ2') -∗
  (Γ1 ++ Γ2 ⊨ (e1 ||| e2) : (A1 * A2) ⫤ Γ1' ++ Γ2').
Proof.
  iIntros "#He1 #He2 !>" (σ) "HΓ /=".
  iDestruct (ctx_ltyped_app with "HΓ")
    as "[HΓ1 HΓ2]".
  wp_apply (wp_par with "(He1 HΓ1) (He2 HΓ2)").
  iIntros (w1 w2) "[[HA1 HΓ1'] [HA2 HΓ2']] !>".
  iSplitL "HA1 HA2".
  + iExists w1, w2. by iFrame.
  + iApply ctx_ltyped_app.
Qed.
```

```
Γ1, Γ1', Γ2, Γ2' : ctx Σ
e1, e2 : expr
A1, A2 : ltty Σ
σ : gmap string val
w1, w2 : val
==========================
"He1" : Γ1 ⊨ e1 : A1 ⫤ Γ1'
"He2" : Γ2 ⊨ e2 : A2 ⫤ Γ2'
--------------------------□
"HA1" : A1 w1
"HA2" : A2 w2
--------------------------∗
(A1 × A2) (w1, w2)
```

$$A_1 \times A_2 \triangleq \lambda w.\, \exists w_1, w_2.\, (w = (w_1, w_2)) * \triangleright(A_1\, w_1) * \triangleright(A_2\, w_2)$$

Semantic Term Types – Example proof

Rule:

$$\frac{\Gamma_1 \vDash e_1 : A_1 \dashv \Gamma_1' \qquad \Gamma_2 \vDash e_2 : A_2 \dashv \Gamma_2'}{\Gamma_1 \cdot \Gamma_2 \vDash (e_1 \mid\mid e_2) : (A_1 \times A_2) \dashv \Gamma_1' \cdot \Gamma_2'}$$

Proof:

```
Lemma ltyped_par Γ1 Γ1' Γ2 Γ2' e1 e2 A1 A2 :
  (Γ1 ⊨ e1 : A1 ⫤ Γ1') -∗ (Γ2 ⊨ e2 : A2 ⫤ Γ2') -∗
  (Γ1 ++ Γ2 ⊨ (e1 ||| e2) : (A1 * A2) ⫤ Γ1' ++ Γ2').
Proof.
  iIntros "#He1 #He2 !>" (σ) "HΓ /=".
  iDestruct (ctx_ltyped_app with "HΓ")
    as "[HΓ1 HΓ2]".
  wp_apply (wp_par with "(He1 HΓ1) (He2 HΓ2)").
  iIntros (w1 w2) "[[HA1 HΓ1'] [HA2 HΓ2']] !>".
  iSplitL "HA1 HA2".
  + iExists w1, w2. by iFrame.
  + iApply ctx_ltyped_app. by iFrame.
Qed.
```

```
Γ1, Γ1', Γ2, Γ2' : ctx Σ
e1, e2 : expr
A1, A2 : ltty Σ
σ : gmap string val
w1, w2 : val
==========================
"He1" : Γ1 ⊨ e1 : A1 ⫤ Γ1'
"He2" : Γ2 ⊨ e2 : A2 ⫤ Γ2'
--------------------------□
"HΓ1'" : Γ1' ⊨ σ
"HΓ2'" : Γ2' ⊨ σ
--------------------------∗
Γ1' ++ Γ2' ⊨ σ
```

Semantic Term Types – Example proof

Rule:

$$\frac{\Gamma_1 \vDash e_1 : A_1 \dashv \Gamma_1' \qquad \Gamma_2 \vDash e_2 : A_2 \dashv \Gamma_2'}{\Gamma_1 \cdot \Gamma_2 \vDash (e_1 \mid\mid e_2) : (A_1 \times A_2) \dashv \Gamma_1' \cdot \Gamma_2'}$$

Proof:

```
Lemma ltyped_par Γ1 Γ1' Γ2 Γ2' e1 e2 A1 A2 :
  (Γ1 ⊨ e1 : A1 ⫤ Γ1') -∗ (Γ2 ⊨ e2 : A2 ⫤ Γ2') -∗
  (Γ1 ++ Γ2 ⊨ (e1 ||| e2) : (A1 * A2) ⫤ Γ1' ++ Γ2').
Proof.
  iIntros "#He1 #He2 !>" (σ) "HΓ /=".
  iDestruct (ctx_ltyped_app with "HΓ")
    as "[HΓ1 HΓ2]".
  wp_apply (wp_par with "(He1 HΓ1) (He2 HΓ2)").
  iIntros (w1 w2) "[[HA1 HΓ1'] [HA2 HΓ2']] !>".
  iSplitL "HA1 HA2".
  + iExists w1, w2. by iFrame.
  + iApply ctx_ltyped_app. by iFrame.
Qed.
```

```
Γ1, Γ1', Γ2, Γ2' : ctx Σ
e1, e2 : expr
A1, A2 : ltty Σ
σ : gmap string val
w1, w2 : val
==========================
"He1" : Γ1 ⊨ e1 : A1 ⫤ Γ1'
"He2" : Γ2 ⊨ e2 : A2 ⫤ Γ2'
--------------------------□
"HΓ1'" : Γ1' ⊨ σ
"HΓ2'" : Γ2' ⊨ σ
--------------------------∗
Γ1' ++ Γ2' ⊨ σ
```

$$\frac{\Gamma_1 \cdot \Gamma_2 \vDash \sigma}{(\Gamma_1 \vDash \sigma) * (\Gamma_2 \vDash \sigma)}$$

Rule:

$$\frac{\Gamma_1 \vDash e_1 : A_1 \dashv \Gamma_1' \qquad \Gamma_2 \vDash e_2 : A_2 \dashv \Gamma_2'}{\Gamma_1 \cdot \Gamma_2 \vDash (e_1 \mid\mid e_2) : (A_1 \times A_2) \dashv \Gamma_1' \cdot \Gamma_2'}$$

Proof:

```
Lemma ltyped_par Γ1 Γ1' Γ2 Γ2' e1 e2 A1 A2 :
  (Γ1 ⊨ e1 : A1 ⫤ Γ1') -∗ (Γ2 ⊨ e2 : A2 ⫤ Γ2') -∗
  (Γ1 ++ Γ2 ⊨ (e1 ||| e2) : (A1 * A2) ⫤ Γ1' ++ Γ2').
Proof.
  iIntros "#He1 #He2 !>" (σ) "HΓ /=".
  iDestruct (ctx_ltyped_app with "HΓ")
    as "[HΓ1 HΓ2]".
  wp_apply (wp_par with "(He1 HΓ1) (He2 HΓ2)").
  iIntros (w1 w2) "[[HA1 HΓ1'] [HA2 HΓ2']] !>".
  iSplitL "HA1 HA2".
  + iExists w1, w2. by iFrame.
  + iApply ctx_ltyped_app. by iFrame.
Qed.
```

```
No more subgoals.
```

Semantic Session Type System

Language

ML-like language

$$e \in \text{Expr} ::= v \mid x \mid \texttt{rec}\, f\ x := e \mid e_1(e_2) \mid$$

Language

ML-like language extended with state

$$
\begin{aligned}
e \in \mathrm{Expr} ::= {} & v \mid x \mid \texttt{rec}\ f\ x := e \mid e_1(e_2) \mid \\
& \texttt{ref}\ (e) \mid {!}\,e \mid e_1 \leftarrow e_2 \mid && \text{(state)}
\end{aligned}
$$

Language

ML-like language extended with state, concurrency

$$
\begin{aligned}
e \in \mathrm{Expr} ::= {} & v \mid x \mid \texttt{rec}\, f\ x := e \mid e_1(e_2) \mid \\
& \texttt{ref}\ (e) \mid !\, e \mid e_1 \leftarrow e_2 \mid && \text{(state)} \\
& e_1 \mid\mid e_2 \mid \texttt{fork}\ \{e\} \mid && \text{(concurrency)}
\end{aligned}
$$

Language

ML-like language extended with state, concurrency, locks

$$
\begin{aligned}
e \in \mathrm{Expr} ::= {} & v \mid x \mid \texttt{rec}\, f\ x := e \mid e_1(e_2) \mid \\
& \texttt{ref}\ (e) \mid !\, e \mid e_1 \leftarrow e_2 \mid && \text{(state)} \\
& e_1 \mid\mid e_2 \mid \texttt{fork}\ \{e\} \mid && \text{(concurrency)} \\
& \texttt{new\_lock}\ () \mid \texttt{acquire}\ e \mid \texttt{release}\ e \mid && \text{(locks)}
\end{aligned}
$$

Language

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

$$
\begin{aligned}
e \in \mathrm{Expr} ::= {} & v \mid x \mid \mathtt{rec}\, f\ x := e \mid e_1(e_2) \mid \\
& \mathtt{ref}\ (e) \mid !\, e \mid e_1 \leftarrow e_2 \mid && \text{(state)} \\
& e_1 \mid\mid e_2 \mid \mathtt{fork}\ \{e\} \mid && \text{(concurrency)} \\
& \mathtt{new\_lock}\ () \mid \mathtt{acquire}\ e \mid \mathtt{release}\ e \mid && \text{(locks)} \\
& \mathtt{new\_chan}\ () \mid \mathtt{send}\ e_1\ e_2 \mid \mathtt{recv}\ e \mid \ldots && \text{(message passing)}
\end{aligned}
$$

Language

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

$$
\begin{aligned}
e \in \mathrm{Expr} ::= {} & v \mid x \mid \texttt{rec}\ f\ x := e \mid e_1(e_2) \mid \\
& \texttt{ref}\ (e) \mid {!}\, e \mid e_1 \leftarrow e_2 \mid && \text{(state)} \\
& e_1 \mid\mid e_2 \mid \texttt{fork}\ \{e\} \mid && \text{(concurrency)} \\
& \texttt{new\_lock}\ () \mid \texttt{acquire}\ e \mid \texttt{release}\ e \mid && \text{(locks)} \\
& \texttt{new\_chan}\ () \mid \texttt{send}\ e_1\ e_2 \mid \texttt{recv}\ e \mid \ldots && \text{(message passing)}
\end{aligned}
$$

Message-passing is:

- **Binary:** Each channel have one pair of endpoints

Language

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

$$
\begin{aligned}
e \in \text{Expr} ::= {} & v \mid x \mid \texttt{rec}\ f\ x := e \mid e_1(e_2) \mid \\
& \texttt{ref}\ (e) \mid !\, e \mid e_1 \leftarrow e_2 \mid && \text{(state)} \\
& e_1 \mid\mid e_2 \mid \texttt{fork}\ \{e\} \mid && \text{(concurrency)} \\
& \texttt{new\_lock}\ () \mid \texttt{acquire}\ e \mid \texttt{release}\ e \mid && \text{(locks)} \\
& \texttt{new\_chan}\ () \mid \texttt{send}\ e_1\ e_2 \mid \texttt{recv}\ e \mid \ldots && \text{(message passing)}
\end{aligned}
$$

Message-passing is:

- **Binary:** Each channel have one pair of endpoints
- **Asynchronous:** `send` does not block, two buffers per endpoint pair

Language

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

$$
\begin{aligned}
e \in \mathrm{Expr} ::= {} & v \mid x \mid \texttt{rec}\ f\ x := e \mid e_1(e_2) \mid \\
& \texttt{ref}\ (e) \mid !\, e \mid e_1 \leftarrow e_2 \mid && \text{(state)} \\
& e_1 \mid\mid e_2 \mid \texttt{fork}\ \{e\} \mid && \text{(concurrency)} \\
& \texttt{new\_lock}\ () \mid \texttt{acquire}\ e \mid \texttt{release}\ e \mid && \text{(locks)} \\
& \texttt{new\_chan}\ () \mid \texttt{send}\ e_1\ e_2 \mid \texttt{recv}\ e \mid \ldots && \text{(message passing)}
\end{aligned}
$$

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 are garbage collected

Semantic Session Types

Session types as a new type kind:

$$\begin{aligned} \mathsf{Type}_{\blacklozenge} &\triangleq ? \\ \mathbf{!}A.\, S &\triangleq ? \\ \mathbf{?}A.\, S &\triangleq ? \\ \texttt{end} &\triangleq ? \end{aligned}$$

Semantic Session Types

Session types as a new type kind:

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

Semantic Session Types

Session types as a new type kind:

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

Needs to capture:

- **Exclusivity** of channel endpoint ownership

Semantic Session Types

Session types as a new type kind:

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

Needs to capture:

- **Exclusivity** of channel endpoint ownership
- **Delegation** of resources

Actris - Dependent Separation Protocols (iProto)

Session type-inspired protocols for functional correctness

Actris - Dependent Separation Protocols (iProto)

Session type-inspired protocols for functional correctness, describing exchanges of:

- Logical variables

Actris - Dependent Separation Protocols (iProto)

Session type-inspired protocols for functional correctness, describing exchanges of:

- Logical variables
- Physical values

Actris - Dependent Separation Protocols (iProto)

Session type-inspired protocols for functional correctness, describing exchanges of:

- Logical variables
- Physical values
- Propositions / ownership

Actris - Dependent Separation Protocols (iProto)

Session type-inspired protocols for functional correctness, describing exchanges of:

- Logical variables
- Physical values
- Propositions / ownership

	Dependent separation protocols	**Session types**
Example	$\mathbf{?}(x:\mathbb{Z})\,\langle x\rangle\{\mathrm{True}\}.\,\mathbf{!}\,(y:\mathbb{Z})\,\langle y\rangle\{y = x+2\}.\,\mathbf{end}$	$\mathbf{?Z}.\,\mathbf{!Z}.\,\mathtt{end}$
Usage	$c \rightarrowtail \mathit{prot}$	$c : \mathtt{chan}\ S$

Actris - Dependent Separation Protocols (iProto)

Session type-inspired protocols for functional correctness, describing exchanges of:

- Logical variables
- Physical values
- Propositions / ownership

	Dependent separation protocols	Session types
Example	$\textbf{?}(x : \mathbb{Z})\,\langle x \rangle\{\mathrm{True}\}.\,\textbf{!}\,(y : \mathbb{Z})\,\langle y \rangle\{y = x + 2\}.\,\textbf{end}$	$\textbf{?Z}.\,\textbf{!Z}.\,\texttt{end}$
Usage	$c \rightarrowtail \mathit{prot}$	$c\ :\ \texttt{chan}\ S$

Program example:

$$\lambda c.\,\texttt{let}\,x := \texttt{recv}\ c\,\texttt{in}\,\texttt{send}\ c\ (x + 2)$$

Actris - Dependent Separation Protocols (iProto)

Session type-inspired protocols for functional correctness, describing exchanges of:

- Logical variables
- Physical values
- Propositions / ownership

	Dependent separation protocols	**Session types**
Example	$\mathbf{?}(x:\mathbb{Z})\,\langle x\rangle\{\mathrm{True}\}.\,\mathbf{!}\,(y:\mathbb{Z})\,\langle y\rangle\{y = x+2\}.\,\mathbf{end}$	$\mathbf{?Z}.\,\mathbf{!Z}.\,\mathtt{end}$
Usage	$c \rightarrowtail \mathit{prot}$	$c : \mathtt{chan}\ S$

Program example:

$$\begin{array}{l}(c \rightarrowtail \mathbf{?}(x:\mathbb{Z})\,\langle x\rangle\{\mathrm{True}\}.\,\mathbf{!}\,(y:\mathbb{Z})\,\langle y\rangle\{y = x+2\}.\,\mathbf{end}) \mathrel{-\!\!*} \\ \mathtt{wp}\ (\lambda c.\,\mathtt{let}\ x := \mathtt{recv}\ c\ \mathtt{in}\ \mathtt{send}\ c\ (x+2))\ \{\mathrm{True}\}\end{array}$$

Semantic Session Types

Session types as dependent separation protocols:

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

Dependent separation protocols: **Example:** $\mathbf{?}(x:\mathbb{Z})\,\langle x\rangle\{\mathrm{True}\}.\,\mathbf{!}\,(y:\mathbb{Z})\,\langle y\rangle\{y = x+2\}.\,\mathbf{end}$ **Usage:** $c \rightarrowtail \mathit{prot}$

Semantic Session Types

Session types as dependent separation protocols:

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

Dependent separation protocols:

Example: $\mathbf{?}(x:\mathbb{Z})\,\langle x\rangle\{\text{True}\}.\,\mathbf{!}\,(y:\mathbb{Z})\,\langle y\rangle\{y = x+2\}.\,\mathbf{end}$

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

Semantic Session Types

Session types as dependent separation protocols:

$$\begin{aligned} \mathrm{Type}_{\blacklozenge} &\triangleq \mathrm{iProto} \\ \mathbf{!}A.\,S &\triangleq \mathbf{!}\,(v : \mathrm{Val})\,\langle v\rangle\{A\,v\}.\,S \\ \mathbf{?}A.\,S &\triangleq \mathbf{?}(v : \mathrm{Val})\,\langle v\rangle\{A\,v\}.\,S \\ \mathtt{end} &\triangleq \mathbf{end} \end{aligned}$$

$$\begin{aligned} \mathrm{Type}_{\bigstar} &\triangleq \mathrm{Val} \to \mathrm{iProp} \\ \mathtt{chan}\ S &\triangleq \lambda w.\,? \end{aligned}$$

Dependent separation protocols:

Example:	$\mathbf{?}(x:\mathbb{Z})\,\langle x\rangle\{\mathrm{True}\}.\,\mathbf{!}\,(y:\mathbb{Z})\,\langle y\rangle\{y = x+2\}.\,\mathbf{end}$
Usage:	$c \rightarrowtail \mathit{prot}$

Semantic Session Types

Session types as dependent separation protocols:

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

Dependent separation protocols:

Example: $\mathbf{?}(x:\mathbb{Z})\,\langle x\rangle\{\mathrm{True}\}.\,\mathbf{!}\,(y:\mathbb{Z})\,\langle y\rangle\{y = x+2\}.\,\mathbf{end}$

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

Rule:

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

Proof:

```
Lemma ltyped_recv Γ x A S :
  Γ !! x = Some (chan (<??> TY A; S))%lty →
  Γ ⊨ recv x : A ⫤ ctx_cons x (chan S) Γ.
Proof.
  iIntros (HΓx%ctx_lookup_perm) "!>".
  iIntros (σ) "HΓ /=". rewrite {1}HΓx /=.
  iDestruct (ctx_ltyped_cons with "HΓ") as
    (c Hσ) "[Hc HΓ]".
  rewrite Hσ.
  wp_recv (v) as "HA".
  iFrame "HA".
  iApply ctx_ltyped_cons; eauto with iFrame.
Qed.
```

Semantic Session Types – Example Proof

Rule:

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

Proof:

```
Lemma ltyped_recv Γ x A S :
  Γ !! x = Some (chan (<??> TY A; S))%lty →
  Γ ⊨ recv x : A ⫤ ctx_cons x (chan S) Γ.
Proof.
  iIntros (HΓx%ctx_lookup_perm) "!>".
  iIntros (σ) "HΓ /=". rewrite {1}HΓx /=.
  iDestruct (ctx_ltyped_cons with "HΓ") as
    (c Hσ) "[Hc HΓ]".
  rewrite Hσ.
  wp_recv (v)
  iFrame "HA"
  iApply ctx_ltyped_cons; eauto with iFrame.
Qed.
```

$$\forall \sigma. (\Gamma, x : \texttt{chan}\ (?A.\ S) \vDash \sigma) \twoheadrightarrow$$
$$\quad \mathsf{wp}\ (\texttt{recv}\ x)[\sigma]\ \{w.\ (A\ w) * (\Gamma, x : \texttt{chan}\ S \vDash \sigma)\}$$

Semantic Session Types – Example Proof

Rule:

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

Proof:

```
Lemma ltyped_recv Γ x A S :
  Γ !! x = Some (chan (<??> TY A; S))%lty →
  Γ ⊨ recv x : A ⫤ ctx_cons x (chan S) Γ.
Proof.
  iIntros (HΓx%ctx_lookup_perm) "!>".
  iIntros (σ) "HΓ /=". rewrite {1}HΓx /=.
  iDestruct (ctx_ltyped_cons with "HΓ") as
    (c Hσ) "[Hc HΓ]".
  rewrite Hσ.
  wp_recv (v) as "HA".
  iFrame "HA".
  iApply ctx_ltyped_cons; eauto with iFrame.
Qed.
```

```
Γ : ctx Σ
x : string
A : ltty Σ
S : lsty Σ
σ : gmap string val
==============================
"HΓ" : Γ,(x:chan (<??>TY A; S))
         ⊨ σ
------------------------------∗
WP recv (σ(x))
  {{ w, A w ∗
     Γ,(x : chan S) ⊨ σ }}
```

Semantic Session Types – Example Proof

Rule:

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

Proof:

```
Lemma ltyped_recv Γ x A S :
  Γ !! x = Some (chan (<??> TY A; S))%lty →
  Γ ⊨ recv x : A ⫤ ctx_cons x (chan S) Γ.
Proof.
  iIntros (HΓx%ctx_lookup_perm) "!>".
  iIntros (σ) "HΓ /=". rewrite {1}HΓx /=.
  iDestruct (ctx_ltype
    (c Hσ) "[Hc HΓ]".
  rewrite Hσ.
  wp_recv (v) as "HA".
  iFrame "HA".
  iApply ctx_ltyped_cons; eauto with iFrame.
Qed.
```

$$\frac{\Gamma, x : A \vDash \sigma}{\exists v.\, (\sigma(x) = v) * (\Gamma \vDash \sigma) * (A\, v)}$$

```
Γ : ctx Σ
x : string
A : ltty Σ
S : lsty Σ
σ : gmap string val
==============================
"HΓ" : Γ,(x:chan (<??>TY A; S))
        ⊨ σ
------------------------------∗
σ(x))
 w ∗
 : chan S) ⊨ σ }}
```

Semantic Session Types – Example Proof

Rule:

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

Proof:

```
Lemma ltyped_recv Γ x A S :
  Γ !! x = Some (chan (<??> TY A; S))%lty →
  Γ ⊨ recv x : A ⫤ ctx_cons x (chan S) Γ.
Proof.
  iIntros (HΓx%ctx_lookup_perm) "!>".
  iIntros (σ) "HΓ /=". rewrite {1}HΓx /=.
  iDestruct (ctx_ltyped_cons with "HΓ") as
    (c Hσ) "[Hc HΓ]".
  rewrite Hσ.
  wp_recv (v) as "HA".
  iFrame "HA".
  iApply ctx_ltyped_cons; eauto with iFrame.
Qed.
```

```
Γ : ctx Σ
x : string
A : ltty Σ
S : lsty Σ
σ : gmap string val
c : val
Hσ : σ(x) = c
==============================
"Hc" : c ↣ (<??> TY A; S)
"HΓ" : Γ ⊨ σ
------------------------------∗
WP recv c
  {{ w, A w ∗
     Γ,(x : chan S) ⊨ σ }}
```

Rule:

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

Proof:

```
Lemma ltyped_recv Γ x A S :
  Γ !! x = Some (chan (<??> TY A; S))%lty →
Pr
```

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

```
  rewrite Hσ.
  wp_recv (v) as "HA".
  iFrame "HA".
  iApply ctx_ltyped_cons; eauto with iFrame.
Qed.
```

```
Γ : ctx Σ
x : string
A : ltty Σ
S : lsty Σ
σ : gmap string val
c : val
Hσ : σ(x) = c
==============================
"Hc" : c ↣ (<??> TY A; S)
"HΓ" : Γ ⊨ σ
------------------------------∗
WP recv c
  {{ w, A w ∗
     Γ,(x : chan S) ⊨ σ }}
```

Rule:

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

Proof:

```
Lemma ltyped_recv Γ x A S :
  Γ !! x = Some (chan (<??> TY A; S))%lty →
  Γ ⊨ recv x : A ⫤ ctx_cons x (chan S) Γ.
Proof.
  iIntros (HΓx%ctx_lookup_perm) "!>".
  iIntros (σ) "HΓ /=". rewrite {1}HΓx /=.
  iDestruct (ctx_ltyped_cons with "HΓ") as
    (c Hσ) "[Hc HΓ]".
  rewrite Hσ.
  wp_recv (v) as "HA".
  iFrame "HA".
  iApply ctx_ltyped_cons; eauto with iFrame.
Qed.
```

```
Γ : ctx Σ
x : string
A : ltty Σ
S : lsty Σ
σ : gmap string val
c : val
Hσ : σ(x) = c
v : val
==============================
"Hc" : c ↣ S
"HΓ" : Γ ⊨ σ
"HA" : A v
------------------------------∗
A v ∗
Γ,(x : chan S) ⊨ σ
```

Rule:

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

Proof:

```
Lemma ltyped_recv Γ x A S :
  Γ !! x = Some (chan (<??> TY A; S))%lty →
  Γ ⊨ recv x : A ⫤ ctx_cons x (chan S) Γ.
Proof.
  iIntros (HΓx%ctx_lookup_perm) "!>".
  iIntros (σ) "HΓ /=". rewrite {1}HΓx /=.
  iDestruct (ctx_ltyped_cons with "HΓ") as
    (c Hσ) "[Hc HΓ]".
  rewrite Hσ.
  wp_recv (v) as "HA".
  iFrame "HA".
  iApply ctx_ltyped_cons; eauto with iFrame.
Qed.
```

```
Γ : ctx Σ
x : string
A : ltty Σ
S : lsty Σ
σ : gmap string val
c : val
Hσ : σ(x) = c
v : val
==============================
"Hc" : c ↣ S
"HΓ" : Γ ⊨ σ
------------------------------∗
Γ,(x : chan S) ⊨ σ
```

Rule:

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

Proof:

```
Lemma ltyped_recv Γ x A S :
  Γ !! x = Some (chan (<??> TY A; S))%lty →
  Γ ⊨ recv x : A ⫤ ctx_cons x (chan S) Γ.
Proof.
  iIntros (HΓx%ctx_lookup_perm) "!>".
  iIntros (σ) "HΓ /=". rewrite {1}HΓx /=
  iDestr
    (c H
  rewrit
  wp_recv (v) as "HA".
  iFrame "HA".
  iApply ctx_ltyped_cons; eauto with iFrame.
Qed.
```

$$\frac{\Gamma, x : A \vDash \sigma}{\exists v.\, (\sigma(x) = v) * (\Gamma \vDash \sigma) * (A\, v)}$$

```
Γ : ctx Σ
x : string
A : ltty Σ
S : lsty Σ
σ : gmap string val
c : val
Hσ : σ(x) = c
v : val
==============================
"Hc" : c ↣ S
"HΓ" : Γ ⊨ σ
------------------------------∗
Γ,(x : chan S) ⊨ σ
```

Semantic Session Types – Example Proof

Rule:

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

Proof:

```
Lemma ltyped_recv Γ x A S :
  Γ !! x = Some (chan (<??> TY A; S))%lty →
  Γ ⊨ recv x : A ⫤ ctx_cons x (chan S) Γ.
Proof.
  iIntros (HΓx%ctx_lookup_perm) "!>".
  iIntros (σ) "HΓ /=". rewrite {1}HΓx /=.
  iDestruct (ctx_ltyped_cons with "HΓ") as
    (c Hσ) "[Hc HΓ]".
  rewrite Hσ.
  wp_recv (v) as "HA".
  iFrame "HA".
  iApply ctx_ltyped_cons; eauto with iFrame.
Qed.
```

```
No more subgoals.
```

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

Product types	Separation conjunction ($*$)

Overview of Features

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

Product types	Separation conjunction ($*$)
Unique references	Points-to connective ($\ell \mapsto v$)

Overview of Features

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

Product types	Separation conjunction ($*$)
Unique references	Points-to connective ($\ell \mapsto v$)
Session types	Actris dependent separation protocols ($c \rightarrowtail S$)

Overview of Features

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

Product types	Separation conjunction ($*$)
Unique references	Points-to connective ($\ell \mapsto v$)
Session types	Actris dependent separation protocols ($c \rightarrowtail S$)
Function types	Wand ($-\!\!*$) and weakest precondition ($\mathsf{wp}\ e\ \{\Phi\}$)

Overview of Features

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

Product types	Separation conjunction ($*$)
Unique references	Points-to connective ($\ell \mapsto v$)
Session types	Actris dependent separation protocols ($c \rightarrowtail S$)
Function types	Wand ($-\!*$) and weakest precondition ($\mathrm{wp}\ e\ \{\Phi\}$)
Shared references	Invariants ($\boxed{P}$)

Overview of Features

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

Product types	Separation conjunction ($*$)
Unique references	Points-to connective ($\ell \mapsto v$)
Session types	Actris dependent separation protocols ($c \rightarrowtail S$)
Function types	Wand ($-\!*$) and weakest precondition ($\mathrm{wp}\ e\ \{\Phi\}$)
Shared references	Invariants ($\boxed{P}$)
Copyable types	Persistent modality ($\square\, P$)

Overview of Features

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

Product types	Separation conjunction ($*$)
Unique references	Points-to connective ($\ell \mapsto v$)
Session types	Actris dependent separation protocols ($c \rightarrowtail S$)
Function types	Wand ($-\!\!*$) and weakest precondition ($\mathrm{wp}\ e\ \{\Phi\}$)
Shared references	Invariants ($\boxed{P}$)
Copyable types	Persistent modality ($\Box\, P$)
Lock types	Iris's lock library

Overview of Features

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

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

Overview of Features

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

Product types	Separation conjunction ($*$)
Unique references	Points-to connective ($\ell \mapsto v$)
Session types	Actris dependent separation protocols ($c \rightarrowtail S$)
Function types	Wand ($-\!\!*$) and weakest precondition ($\mathrm{wp}\ e\ \{\Phi\}$)
Shared references	Invariants ($\boxed{P}$)
Copyable types	Persistent modality ($\square\, P$)
Lock types	Iris's lock library
Session choice types	Actris dependent separation protocols ($c \rightarrowtail S$)
Recursion	Guarded step-indexed recursion ($\triangleright$, μ)

Overview of Features

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

Product types	Separation conjunction ($*$)
Unique references	Points-to connective ($\ell \mapsto v$)
Session types	Actris dependent separation protocols ($c \rightarrowtail S$)
Function types	Wand ($-\!\!*$) and weakest precondition ($\mathrm{wp}\ e\ \{\Phi\}$)
Shared references	Invariants ($\boxed{P}$)
Copyable types	Persistent modality ($\Box\, P$)
Lock types	Iris's lock library
Session choice types	Actris dependent separation protocols ($c \rightarrowtail S$)
Recursion	Guarded step-indexed recursion ($\triangleright$, μ)
Term polymorphism	Higher-order impredicative quantification ($\forall$, $\exists$)

Overview of Features

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

Product types	Separation conjunction ($*$)
Unique references	Points-to connective ($\ell \mapsto v$)
Session types	Actris dependent separation protocols ($c \rightarrowtail S$)
Function types	Wand ($-\!\!*$) and weakest precondition ($\mathrm{wp}\ e\ \{\Phi\}$)
Shared references	Invariants ($\boxed{P}$)
Copyable types	Persistent modality ($\Box\, P$)
Lock types	Iris's lock library
Session choice types	Actris dependent separation protocols ($c \rightarrowtail S$)
Recursion	Guarded step-indexed recursion ($\triangleright$, μ)
Term polymorphism	Higher-order impredicative quantification ($\forall$, $\exists$)
Session polymorphism	Higher-order impredicative protocols binders

Overview of Features

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

Product types	Separation conjunction ($*$)
Unique references	Points-to connective ($\ell \mapsto v$)
Session types	Actris dependent separation protocols ($c \rightarrowtail S$)
Function types	Wand ($-\!\!*$) and weakest precondition ($\mathrm{wp}\ e\ \{\Phi\}$)
Shared references	Invariants ($\boxed{P}$)
Copyable types	Persistent modality ($\square P$)
Lock types	Iris's lock library
Session choice types	Actris dependent separation protocols ($c \rightarrowtail S$)
Recursion	Guarded step-indexed recursion ($\triangleright$, μ)
Term polymorphism	Higher-order impredicative quantification ($\forall$, $\exists$)
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

Product types	Separation conjunction ($*$)
Unique references	Points-to connective ($\ell \mapsto v$)
Session types	Actris dependent separation protocols ($c \rightarrowtail S$)
Function types	Wand ($-\!\!*$) and weakest precondition ($\mathrm{wp}\ e\ \{\Phi\}$)
Shared references	Invariants ($\boxed{P}$)
Copyable types	Persistent modality ($\Box P$)
Lock types	Iris's lock library
Session choice types	Actris dependent separation protocols ($c \rightarrowtail S$)
Recursion	Guarded step-indexed recursion ($\triangleright$, μ)
Term polymorphism	Higher-order impredicative quantification ($\forall$, $\exists$)
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

Function types: $A \multimap B \triangleq \lambda w.\, \forall v.\, \triangleright(A\, v) -\!\!*\ \mathsf{wp}\ (w\ v)\ \{B\}$

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

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

Lock types:

$$\mathtt{mutex}\, A \triangleq \lambda w.\, \exists lk, \ell.\, (w = (lk, \ell)) * \mathsf{is\_lock}\ lk\ (\exists v.\, (\ell \mapsto u) * \triangleright(A\, v))$$
$$\overline{\mathtt{mutex}}\, A \triangleq \lambda w.\, \exists lk, \ell.\, (w = (lk, \ell)) * \mathsf{is\_lock}\ lk\ (\exists v.\, (\ell \mapsto u) * \triangleright(A\, v)) * (\ell \mapsto -)$$

Session choice:

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

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

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

Manual Typing Proofs

An Untypeable Program

Recall the following example:

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

An Untypeable Program

Recall the following example:

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

An Untypeable Program

Recall the following example:

$$\vDash \lambda c.\, (\texttt{recv}\ c \mathbin{||} \texttt{recv}\ c) : \texttt{chan}\ (\mathbf{?Z}.\, \mathbf{?Z}.\, \mathrm{end}) \multimap (\mathbf{Z} \times \mathbf{Z}) \quad \checkmark$$

An Untypeable Program

Recall the following example:

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

The rule is just another lemma

An Untypeable Program

Recall the following example:

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

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

$$(c \rightarrowtail \mathbf{?}(v_1 : \mathsf{Val})\, \langle v_1 \rangle \{v_1 \in \mathbb{Z}\}.\, \mathbf{?}(v_2 : \mathsf{Val})\, \langle v_2 \rangle \{v_2 \in \mathbb{Z}\}.\, \mathbf{end}) \twoheadrightarrow$$
$$\mathsf{wp}\ (\texttt{recv}\ c \mid\mid \texttt{recv}\ c)\ \{w.\, \exists w_1, w_2.\, (w = (w_1, w_2)) * \triangleright(w_1 \in \mathbb{Z}) * \triangleright(w_2 \in \mathbb{Z})\}$$

An Untypeable Program

Recall the following example:

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

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

$$(c \rightarrowtail \mathbf{?}(v_1 : \mathsf{Val})\,\langle v_1 \rangle \{v_1 \in \mathbb{Z}\}.\,\mathbf{?}(v_2 : \mathsf{Val})\,\langle v_2 \rangle \{v_2 \in \mathbb{Z}\}.\,\mathbf{end}) \mathrel{-\!\!*}$$
$$\mathsf{wp}\ (\texttt{recv}\ c \parallel \texttt{recv}\ c)\ \{w.\,\exists w_1, w_2.\,(w = (w_1, w_2)) * \triangleright(w_1 \in \mathbb{Z}) * \triangleright(w_2 \in \mathbb{Z})\}$$

An Untypeable Program

Recall the following example:

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

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

$$(c \rightarrowtail \mathbf{?}(v_1 : \mathsf{Val})\, \langle v_1 \rangle \{v_1 \in \mathbb{Z}\}.\, \mathbf{?}(v_2 : \mathsf{Val})\, \langle v_2 \rangle \{v_2 \in \mathbb{Z}\}.\, \mathbf{end}) \mathrel{-\!\!*}$$
$$\mathsf{wp}\, (\texttt{recv}\ c \mid\mid \texttt{recv}\ c)\, \{w.\, \exists w_1, w_2.\, (w = (w_1, w_2)) * \triangleright(w_1 \in \mathbb{Z}) * \triangleright(w_2 \in \mathbb{Z})\}$$

An Untypeable Program

Recall the following example:

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

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

$$(c \rightarrowtail \mathbf{?}(v_1 : \mathrm{Val})\,\langle v_1\rangle\{v_1 \in \mathbb{Z}\}.\,\mathbf{?}(v_2 : \mathrm{Val})\,\langle v_2\rangle\{v_2 \in \mathbb{Z}\}.\,\mathbf{end}) \twoheadrightarrow$$
$$\mathrm{wp}\,(\texttt{recv}\ c \parallel \texttt{recv}\ c)\,\{w.\,\exists w_1, w_2.\,(w = (w_1, w_2)) * \triangleright(w_1 \in \mathbb{Z}) * \triangleright(w_2 \in \mathbb{Z})\}$$

And then using Iris's ghost state machinery!

An Untypeable Program

Recall the following example:

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

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

$$(c \rightarrowtail \mathbf{?}(v_1 : \mathsf{Val})\,\langle v_1\rangle\{v_1 \in \mathbb{Z}\}.\,\mathbf{?}(v_2 : \mathsf{Val})\,\langle v_2\rangle\{v_2 \in \mathbb{Z}\}.\,\mathbf{end}) \mathrel{-\!\!*}$$
$$\mathsf{wp}\,(\texttt{recv}\ c \parallel \texttt{recv}\ c)\,\{w.\,\exists w_1, w_2.\,(w = (w_1, w_2)) * \triangleright(w_1 \in \mathbb{Z}) * \triangleright(w_2 \in \mathbb{Z})\}$$

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

Concluding Remarks

Summary

1. **Feature-rich session type systems**
   - We combined polymorphism, recursion, (asynchronous) subtyping, and more
   - By exploiting the expressivity of Iris and Actris

Summary

1. **Feature-rich session type systems**
   - We combined polymorphism, recursion, (asynchronous) subtyping, and more
   - By exploiting the expressivity of Iris and Actris
2. **Support for "racy" yet safe programs**
   - We extend the type system with judgments for "racy" programs like
     $$\Gamma \vDash \lambda c.\,(\texttt{recv}\ c \parallel \texttt{recv}\ c) : \texttt{chan}\ (?\mathbf{Z}.\,?\mathbf{Z}.\,\texttt{end}) \multimap (\mathbf{Z} \times \mathbf{Z}) \dashv \Gamma$$
   - By unfolding the definitions and using Iris ghost mechanisms

Summary

1. **Feature-rich session type systems**
   - We combined polymorphism, recursion, (asynchronous) subtyping, and more
   - By exploiting the expressivity of Iris and Actris

2. **Support for "racy" yet safe programs**
   - We extend the type system with judgments for "racy" programs like

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

   - By unfolding the definitions and using Iris ghost mechanisms

3. **Mechanised soundness proof of our results**
   - We mechanised it in Coq: https://gitlab.mpi-sws.org/iris/actris/-/tree/cpp21
   - By building on top of Iris and Actris frameworks and libraries
   - Artifact: https://zenodo.org/record/4322752