Mechanized verification of type systems using Iris

Robbert Krebbers

Radboud University Nijmegen, The Netherlands

April 13, 2026 @ IFIP WG 1.9/2.15 Verified Software, Torino, Italy

The old problem of proving “type safety”:

“Well-typed programs cannot go wrong”

The old problem of proving "type safety":

If $\vdash e : A$ then $\mathsf{safe}(e)$

The old problem of proving "type safety":

If $\vdash e : A$ then $\mathrm{safe}(e)$

Goal of this talk:

- Introduce the "logical approach" in separation logic as an alternative to the standard progress/preservation approach to type safety
- Show that this approach is well-suited for mechanization of challenging substructural type systems (*e.g.*, session types and Rust) in Rocq
- Show that this approach makes it possible to type "unsafe" code

Recap: Progress and preservation [Wright and Felleisen, simplified by Harper]

Safety is defined in terms of a small-step operational semantics:

$$\mathsf{safe}(e) \triangleq \forall e'.\, (e \to^* e') \Rightarrow e' \in \mathsf{Val} \vee \mathsf{reducible}(e')$$

Recap: Progress and preservation [Wright and Felleisen, simplified by Harper]

Safety is defined in terms of a small-step operational semantics:

$$\mathsf{safe}(e) \triangleq \forall e'.\,(e \to^* e') \Rightarrow e' \in \mathsf{Val} \vee \mathsf{reducible}(e')$$

1. **Progress:** If $\vdash e : A$ then $e \in \mathsf{Val}$ or $\mathsf{reducible}(e)$
2. **Preservation:** If $\vdash e : A$ and $e \to e'$ then $\vdash e' : A$

Recap: Progress and preservation [Wright and Felleisen, simplified by Harper]

Safety is defined in terms of a small-step operational semantics:

$$\mathsf{safe}(e) \triangleq \forall e'.\,(e \to^* e') \Rightarrow e' \in \mathsf{Val} \vee \mathsf{reducible}(e')$$

1. **Progress:** If $\vdash e : A$ then $e \in \mathsf{Val}$ or $\mathsf{reducible}(e)$
2. **Preservation:** If $\vdash e : A$ and $e \to e'$ then $\vdash e' : A$

Proof of type safety: If $\vdash e : A$ then $\mathsf{safe}(e)$
Obtain $\vdash e' : A$ by induction on length of $e \to^* e'$ and preservation,
conclude by progress

Problems of progress and preservation

Progress and preservation are extremely effective and simple to teach, but:

Problems of progress and preservation

Progress and preservation are extremely effective and simple to teach, but:

- It becomes much more complicated when considering a language with a state

Preservation: If $\Sigma; \Gamma \vdash e : A$ and $\Sigma \vdash_{\mathrm{h}} \sigma$ and $(\sigma, e) \to (\sigma', e')$ then there exists $\Sigma' \supseteq \Sigma$ such that $\Sigma'; \Gamma \vdash e' : A$ and $\Sigma' \vdash_{\mathrm{h}} \sigma'$

Problems of progress and preservation

Progress and preservation are extremely effective and simple to teach, but:

- It becomes much more complicated when considering a language with a state

  Preservation: If $\Sigma; \Gamma \vdash e : A$ and $\Sigma \vdash_{\mathrm{h}} \sigma$ and $(\sigma, e) \to (\sigma', e')$ then there exists $\Sigma' \supseteq \Sigma$ such that $\Sigma'; \Gamma \vdash e' : A$ and $\Sigma' \vdash_{\mathrm{h}} \sigma'$

- Even more tricky once you consider a substructural type system
  Disjointness conditions show up everywhere
  (And Rocq does not accept "left as an exercise for the reader")

Problems of progress and preservation

Progress and preservation are extremely effective and simple to teach, but:

- It becomes much more complicated when considering a language with a state

  Preservation: If $\Sigma; \Gamma \vdash e : A$ and $\Sigma \vdash_{\mathrm{h}} \sigma$ and $(\sigma, e) \to (\sigma', e')$ then there exists $\Sigma' \supseteq \Sigma$ such that $\Sigma'; \Gamma \vdash e' : A$ and $\Sigma' \vdash_{\mathrm{h}} \sigma'$

- Even more tricky once you consider a substructural type system
  Disjointness conditions show up everywhere
  (And Rocq does not accept "left as an exercise for the reader")
- Unsuitable to reason about "unsafe" code
  `unsafe` in Rust, `Obj.magic` in OCaml, `unsafePerformIO` in Haskell

Semantic typing

Define "semantic typing judgment" $\vDash e : A$ in terms of language semantics

Not as an inductive relation!

Semantic typing

Define "semantic typing judgment" $\vDash e : A$ in terms of language semantics
Not as an inductive relation!

1. **Adequacy:** If $\vDash e : A$ implies $\mathsf{safe}(e)$

Semantic typing

Define "semantic typing judgment" $\vDash e : A$ in terms of language semantics
Not as an inductive relation!

1. **Adequacy:** If $\vDash e : A$ implies $\text{safe}(e)$
2. **Fundamental theorem:** If $\vdash e : A$ implies $\vDash e : A$

Semantic typing

Define "semantic typing judgment" $\vDash e : A$ in terms of language semantics
Not as an inductive relation!

1. **Adequacy:** If $\vDash e : A$ implies $\text{safe}(e)$
2. **Fundamental theorem:** If $\vdash e : A$ implies $\vDash e : A$

Proof of type safety: If $\vdash e : A$ then $\text{safe}(e)$
Modus ponens with fundamental theorem and adequacy

Semantic typing

Define "semantic typing judgment" $\vDash e : A$ in terms of language semantics
Not as an inductive relation!

1. **Adequacy:** If $\vDash e : A$ implies $\text{safe}(e)$
   Usually the easy part, since safety is part of the definition of $\vDash e : A$
2. **Fundamental theorem:** If $\vdash e : A$ implies $\vDash e : A$

Proof of type safety: If $\vdash e : A$ then $\text{safe}(e)$
Modus ponens with fundamental theorem and adequacy

Semantic typing

Define "semantic typing judgment" $\vDash e : A$ in terms of language semantics
Not as an inductive relation!

1. **Adequacy:** If $\vDash e : A$ implies $\mathrm{safe}(e)$
   Usually the easy part, since safety is part of the definition of $\vDash e : A$
2. **Fundamental theorem:** If $\vdash e : A$ implies $\vDash e : A$
   Induction on the derivation of $\vdash e : A$

Proof of type safety: If $\vdash e : A$ then $\mathrm{safe}(e)$
Modus ponens with fundamental theorem and adequacy

Semantic typing

Define "semantic typing judgment" $\vDash e : A$ in terms of language semantics
Not as an inductive relation!

1. **Adequacy:** If $\vDash e : A$ implies safe(e)
   Usually the easy part, since safety is part of the definition of $\vDash e : A$
2. **Fundamental theorem:** If $\vdash e : A$ implies $\vDash e : A$
   Induction on the derivation of $\vdash e : A$
   The work is in proving the "compatibility lemmas": semantic versions ($\vDash$) of each syntactic typing rule ($\vdash$)

$$\frac{\vdash e_1 : A \to B \qquad \vdash e_2 : A}{\vdash e_1\ e_2 : B}$$

Proof of type safety: If $\vdash e : A$ then safe(e)
Modus ponens with fundamental theorem and adequacy

Semantic typing

Define "semantic typing judgment" $\vDash e : A$ in terms of language semantics
Not as an inductive relation!

1. **Adequacy:** If $\vDash e : A$ implies $\mathsf{safe}(e)$
   Usually the easy part, since safety is part of the definition of $\vDash e : A$
2. **Fundamental theorem:** If $\vdash e : A$ implies $\vDash e : A$
   Induction on the derivation of $\vdash e : A$
   The work is in proving the "compatibility lemmas": semantic versions ($\vDash$) of each syntactic typing rule ($\vdash$)

$$\frac{\vDash e_1 : A \to B \qquad \vDash e_2 : A}{\vDash e_1\ e_2 : B}$$

Proof of type safety: If $\vdash e : A$ then $\mathsf{safe}(e)$
Modus ponens with fundamental theorem and adequacy

Key challenge: Define $\vDash e : A$ so that:

- It is rich enough to support challenging PL features
- It allows for a concise proof of the fundamental theorem

A bit of history

- Milner's original type safety proof (1978) was a semantic one
- It remained an open challenge for a long time to scale the semantic approach to languages with polymorphism, recursive types, and ML-style references

A bit of history

- Milner's original type safety proof (1978) was a semantic one
- It remained an open challenge for a long time to scale the semantic approach to languages with polymorphism, recursive types, and ML-style references
- A breakthrough was the work on **step-indexing** by Appel, Ahmed and collaborators (2001–2004)

- More abstract versions developed by Appel *et al.* (2007) and Dreyer *et al.* (2011)

A bit of history

- Milner's original type safety proof (1978) was a semantic one
- It remained an open challenge for a long time to scale the semantic approach to languages with polymorphism, recursive types, and ML-style references
- A breakthrough was the work on **step-indexing** by Appel, Ahmed and collaborators (2001–2004)

- More abstract versions developed by Appel *et al.* (2007) and Dreyer *et al.* (2011)
- Iris provides a modern **logical approach** in which concurrent separation logic hides reasoning about state *and* which is well-suited for mechanized proofs in Rocq

In what follows, I will show the simplest semantic proof for simply-typed lambda calculus (STLC)

In what follows, I will show the simplest semantic proof for simply-typed lambda calculus (STLC)

And then change some conjunctions into separation conjunctions to scale to a substructural type system

Semantic typing for STLC

Semantic interpretation of types ("logical relation"):

$$[\![_]\!] : \mathsf{Type} \to \mathsf{SemType} \quad \text{where} \quad \mathsf{SemType} \triangleq \mathsf{Val} \to \mathsf{Prop}$$

Semantic typing for STLC

Semantic interpretation of types ("logical relation"):

$$[\![_]\!] : \mathsf{Type} \to \mathsf{SemType} \quad \text{where} \quad \mathsf{SemType} \triangleq \mathsf{Val} \to \mathsf{Prop}$$

$$[\![\mathbf{Z}]\!] \triangleq \lambda v.\, v \in \mathbb{Z}$$

Semantic typing for STLC

Semantic interpretation of types ("logical relation"):

$$[\![_]\!] : \mathsf{Type} \to \mathsf{SemType} \quad \text{where} \quad \mathsf{SemType} \triangleq \mathsf{Val} \to \mathsf{Prop}$$

$$[\![\mathbf{Z}]\!] \triangleq \lambda v.\, v \in \mathbb{Z}$$

$$[\![A \times B]\!] \triangleq \lambda v.\, \exists v_1, v_2.\, v = (v_1, v_2) \land [\![A]\!] v_1 \land [\![B]\!] v_2$$

Semantic typing for STLC

Semantic interpretation of types ("logical relation"):

$$[\![_]\!] : \mathsf{Type} \to \mathsf{SemType} \quad \text{where} \quad \mathsf{SemType} \triangleq \mathsf{Val} \to \mathsf{Prop}$$

$$[\![\mathbf{Z}]\!] \triangleq \lambda v.\, v \in \mathbb{Z}$$

$$[\![A \times B]\!] \triangleq \lambda v.\, \exists v_1, v_2.\, v = (v_1, v_2) \wedge [\![A]\!]\, v_1 \wedge [\![B]\!]\, v_2$$

$$[\![A \to B]\!] \triangleq \lambda v.\, \forall w.\, [\![A]\!]\, w \Rightarrow \mathsf{wp}\,(v\ w)\,\{[\![B]\!]\}$$

application is not a value, we need to talk about its result

Semantic typing for STLC

Semantic interpretation of types ("logical relation"):

$$\llbracket _ \rrbracket : \mathsf{Type} \to \mathsf{SemType} \quad \text{where} \quad \mathsf{SemType} \triangleq \mathsf{Val} \to \mathsf{Prop}$$

$$\begin{aligned} \llbracket \mathbf{Z} \rrbracket &\triangleq \lambda v.\, v \in \mathbb{Z} \\ \llbracket A \times B \rrbracket &\triangleq \lambda v.\, \exists v_1, v_2.\, v = (v_1, v_2) \wedge \llbracket A \rrbracket v_1 \wedge \llbracket B \rrbracket v_2 \\ \llbracket A \to B \rrbracket &\triangleq \lambda v.\, \forall w.\, \llbracket A \rrbracket w \Rightarrow \mathsf{wp}\,(v\ w)\,\{\llbracket B \rrbracket\} \end{aligned}$$

Weakest precondition:

$$\mathsf{wp}\ _\ \{_\} : \mathsf{Expr} \to (\mathsf{Val} \to \mathsf{Prop}) \to \mathsf{Prop}$$

$$\mathsf{wp}\ e\ \{\Phi\} \triangleq \mathsf{safe}(e) \wedge (\forall v.\, e \to^* v \Rightarrow \Phi\ v)$$

Semantic typing for STLC

Semantic interpretation of types ("logical relation"):

$$[\![_]\!] : \mathsf{Type} \to \mathsf{SemType} \quad \text{where} \quad \mathsf{SemType} \triangleq \mathsf{Val} \to \mathsf{Prop}$$

$$\begin{aligned}
[\![\mathbf{Z}]\!] &\triangleq \lambda v.\, v \in \mathbb{Z} \\
[\![A \times B]\!] &\triangleq \lambda v.\, \exists v_1, v_2.\, v = (v_1, v_2) \wedge [\![A]\!] v_1 \wedge [\![B]\!] v_2 \\
[\![A \to B]\!] &\triangleq \lambda v.\, \forall w.\, [\![A]\!] w \Rightarrow \mathsf{wp}\,(v\ w)\,\{[\![B]\!]\}
\end{aligned}$$

Weakest precondition:

$$\mathsf{wp}\ _\ \{_\} : \mathsf{Expr} \to (\mathsf{Val} \to \mathsf{Prop}) \to \mathsf{Prop}$$

$$\mathsf{wp}\ e\ \{\Phi\} \triangleq \mathsf{safe}(e) \wedge (\forall v.\, e \to^* v \Rightarrow \Phi\ v)$$

Semantic typing judgment:

$$\vDash e : A \triangleq \mathsf{wp}\ e\ \{[\![A]\!]\}$$

Semantic typing for STLC

Semantic interpretation of types ("logical relation"):

$$[\![_]\!] : \mathsf{Type} \to \mathsf{SemType} \quad \text{where} \quad \mathsf{SemType} \triangleq \mathsf{Val} \to \mathsf{Prop}$$

$$\begin{aligned}
[\![\mathbf{Z}]\!] &\triangleq \lambda v.\, v \in \mathbb{Z} \\
[\![A \times B]\!] &\triangleq \lambda v.\, \exists v_1, v_2.\, v = (v_1, v_2) \wedge [\![A]\!] v_1 \wedge [\![B]\!] v_2 \\
[\![A \to B]\!] &\triangleq \lambda v.\, \forall w.\, [\![A]\!] w \Rightarrow \mathsf{wp}\,(v\ w)\,\{[\![B]\!]\}
\end{aligned}$$

Weakest precondition:

$$\mathsf{wp}\ _\ \{_\} : \mathsf{Expr} \to (\mathsf{Val} \to \mathsf{Prop}) \to \mathsf{Prop}$$

closing substitution, I will ignore those most of the time

Semantic typing judgment:

$$\vDash e : A \triangleq \mathsf{wp}\ e\ \{[\![A]\!]\}$$

Semantic typing for STLC

Semantic interpretation of types ("logical relation"):

$$[\![_]\!] : \mathsf{Type} \to \mathsf{SemType} \quad \text{where} \quad \mathsf{SemType} \triangleq \mathsf{Val} \to \mathsf{Prop}$$

$$[\![\mathbf{Z}]\!] \triangleq \lambda v.\, v \in \mathbb{Z}$$

$$[\![A \times B]\!] \triangleq \lambda v.\, \exists v_1, v_2.\, v = (v_1, v_2) \wedge [\![A]\!] v_1 \wedge [\![B]\!] v_2$$

$$[\![A \to B]\!] \triangleq \lambda v.\, \forall w.\, [\![A]\!] w \Rightarrow \mathsf{wp}\ (v\ w)\ \{[\![B]\!]\}$$

Weakest precondition:

$$\mathsf{wp}\ _\ \{_\} : \mathsf{Expr} \to (\mathsf{Val} \to \mathsf{Prop}) \to \mathsf{Prop}$$

$$\mathsf{wp}\ e\ \{\Phi\} \triangleq \mathsf{safe}(e) \wedge (\forall v.\, e \to^* v \Rightarrow \Phi\ v)$$

Semantic typing judgment:

$$\Gamma \vDash e : A \triangleq \forall \gamma.\, [\![\Gamma]\!] \gamma \Rightarrow \mathsf{wp}\ \gamma(e)\ \{[\![A]\!]\}$$

Proofs of key properties

1. **Adequacy:** If $\vDash e : A$ implies $\text{safe}(e)$
2. **Fundamental theorem:** If $\vdash e : A$ implies $\vDash e : A$

Proofs of key properties

1. **Adequacy:** If $\vDash e : A$ implies $\mathrm{safe}(e)$
   Holds by definition $\vDash e : A \;=\; \mathrm{wp}\; e \,\{[\![A]\!]\} \;=\; \mathrm{safe}(e) \wedge \ldots$
2. **Fundamental theorem:** If $\vdash e : A$ implies $\vDash e : A$

Proofs of key properties

1. **Adequacy:** If $\vDash e : A$ implies $\mathsf{safe}(e)$
   Holds by definition $\vDash e : A \; = \; \mathsf{wp}\; e \, \{[\![A]\!]\} \; = \; \mathsf{safe}(e) \wedge \ldots$
2. **Fundamental theorem:** If $\vdash e : A$ implies $\vDash e : A$
   Induction on the derivation of $\vdash e : A$
   The work is in proving a semantic version ($\vDash$) of each syntactic typing rule ($\vdash$)

$$\frac{\vdash e_1 : A \to B \qquad \vdash e_2 : A}{\vdash e_1 \; e_2 : B}$$

Proofs of key properties

1. **Adequacy:** If $\vDash e : A$ implies $\text{safe}(e)$
   Holds by definition $\vDash e : A \; = \; \text{wp}\, e \, \{[\![A]\!]\} \; = \; \text{safe}(e) \wedge \ldots$
2. **Fundamental theorem:** If $\vdash e : A$ implies $\vDash e : A$
   Induction on the derivation of $\vdash e : A$
   The work is in proving a semantic version ($\vDash$) of each syntactic typing rule ($\vdash$)

$$\frac{\vDash e_1 : A \to B \qquad \vDash e_2 : A}{\vDash e_1 \; e_2 : B}$$

The proof is streamlined through abstract reasoning rules about WP, thereby avoiding explicit reasoning abut the operational semantics

An "unsafe" fixpoint combinator

Consider a strict version of Curry's fixpoint operator:

$$\textbf{fix} \triangleq \lambda f.\, (\lambda x.\, f\ (\lambda v.\, x\ x\ v))\ (\lambda x.\, f\ (\lambda v.\, x\ x\ v))$$

An "unsafe" fixpoint combinator

Consider a strict version of Curry's fixpoint operator:

$$\textbf{fix} \triangleq \lambda f.\,(\lambda x.\,f\ (\lambda v.\,x\ x\ v))\ (\lambda x.\,f\ (\lambda v.\,x\ x\ v))$$

Do we have?

$$\vdash \textbf{fix} : \big((A \to B) \to (A \to B)\big) \to (A \to B)$$

An "unsafe" fixpoint combinator

Consider a strict version of Curry's fixpoint operator:

$$\mathbf{fix} \triangleq \lambda f.\,(\lambda x.\, f\ (\lambda v.\, x\ x\ v))\ (\lambda x.\, f\ (\lambda v.\, x\ x\ v))$$

Do we have?

$$\vdash \mathbf{fix} : \big((A \to B) \to (A \to B)\big) \to (A \to B)$$

✗ No. The rules of STLC cannot type check **fix**

An "unsafe" fixpoint combinator

Consider a strict version of Curry's fixpoint operator:

$$\textbf{fix} \triangleq \lambda f.\, (\lambda x.\, f\ (\lambda v.\, x\ x\ v))\ (\lambda x.\, f\ (\lambda v.\, x\ x\ v))$$

Do we have?

$$\vdash \textbf{fix} : \big((A \to B) \to (A \to B)\big) \to (A \to B)$$

✗ No. The rules of STLC cannot type check **fix**

Do we have?

$$\vDash \textbf{fix} : \big((A \to B) \to (A \to B)\big) \to (A \to B)$$

An "unsafe" fixpoint combinator

Consider a strict version of Curry's fixpoint operator:

$$\textbf{fix} \triangleq \lambda f.\, (\lambda x.\, f\ (\lambda v.\, x\ x\ v))\ (\lambda x.\, f\ (\lambda v.\, x\ x\ v))$$

Do we have?

$$\vdash \textbf{fix} : \big((A \to B) \to (A \to B)\big) \to (A \to B)$$

✗ No. The rules of STLC cannot type check **fix**

Do we have?

$$\vDash \textbf{fix} : \big((A \to B) \to (A \to B)\big) \to (A \to B)$$

✓ Yes. We can prove that **fix** is semantically safe

A more feature-rich logical relation

$$[\![_]\!]_\delta : \mathsf{Type} \to \mathsf{SemType} \quad \text{where} \quad \mathsf{SemType} \triangleq \mathsf{Val} \to \mathsf{Prop}$$

$$[\![\mathbf{Z}]\!]_\delta \triangleq \lambda v.\, v \in \mathbb{Z}$$

$$[\![A \times B]\!]_\delta \triangleq \lambda v.\, \exists v_1, v_2.\, v = (v_1, v_2) \land [\![A]\!]_\delta\, v_1 \land [\![B]\!]_\delta\, v_2$$

$$[\![A \to B]\!]_\delta \triangleq \lambda v.\, \forall w.\, [\![A]\!]_\delta\, w \Rightarrow \mathsf{wp}\,(v\ \ w)\,\{[\![B]\!]_\delta\}$$

$$[\![\forall X.\, A]\!]_\delta \triangleq \lambda v.\, \forall \Phi : \mathsf{SemType}.\, \mathsf{wp}\,(v\langle\rangle)\,\{[\![A]\!]_{\delta, X \mapsto \Phi}\}$$

$$[\![\exists X.\, A]\!]_\delta \triangleq \lambda v.\, \exists \Phi : \mathsf{SemType}.\, \exists w.\, v = \mathtt{pack}\langle w\rangle \land [\![A]\!]_{\delta, X \mapsto \Phi}\, w$$

A more feature-rich logical relation

$$\llbracket _ \rrbracket_\delta : \mathsf{Type} \to \mathsf{SemType} \quad \text{where} \quad \mathsf{SemType} \triangleq \mathsf{Val} \to \mathsf{Prop}$$

$$\begin{aligned}
\llbracket \mathbf{Z} \rrbracket_\delta &\triangleq \lambda v.\, v \in \mathbb{Z} \\
\llbracket A \times B \rrbracket_\delta &\triangleq \lambda v.\, \exists v_1, v_2.\, v = (v_1, v_2) \wedge \llbracket A \rrbracket_\delta\, v_1 \wedge \llbracket B \rrbracket_\delta\, v_2 \\
\llbracket A \to B \rrbracket_\delta &\triangleq \lambda v.\, \forall w.\, \llbracket A \rrbracket_\delta\, w \Rightarrow \mathsf{wp}\,(v\ w)\,\{\llbracket B \rrbracket_\delta\} \\
\llbracket \forall X.\, A \rrbracket_\delta &\triangleq \lambda v.\, \forall \Phi : \mathsf{SemType}.\, \mathsf{wp}\,(v\langle\rangle)\,\{\llbracket A \rrbracket_{\delta, X \mapsto \Phi}\} \\
\llbracket \exists X.\, A \rrbracket_\delta &\triangleq \lambda v.\, \exists \Phi : \mathsf{SemType}.\, \exists w.\, v = \mathtt{pack}\langle w \rangle \wedge \llbracket A \rrbracket_{\delta, X \mapsto \Phi}\, w
\end{aligned}$$

Important observations:

- Each type is given a semantic interpretation via the Curry-Howard correspondence

A more feature-rich logical relation

$$\llbracket _ \rrbracket_\delta : \mathsf{Type} \to \mathsf{SemType} \quad \text{where} \quad \mathsf{SemType} \triangleq \mathsf{Val} \to \mathsf{Prop}$$

$$\begin{aligned}
\llbracket \mathbf{Z} \rrbracket_\delta &\triangleq \lambda v.\, v \in \mathbb{Z} \\
\llbracket A \times B \rrbracket_\delta &\triangleq \lambda v.\, \exists v_1, v_2.\, v = (v_1, v_2) \land \llbracket A \rrbracket_\delta\, v_1 \land \llbracket B \rrbracket_\delta\, v_2 \\
\llbracket A \to B \rrbracket_\delta &\triangleq \lambda v.\, \forall w.\, \llbracket A \rrbracket_\delta\, w \Rightarrow \mathsf{wp}\,(v\ w)\,\{\llbracket B \rrbracket_\delta\} \\
\llbracket \forall X.\, A \rrbracket_\delta &\triangleq \lambda v.\, \forall \Phi : \mathsf{SemType}.\, \mathsf{wp}\,(v\langle\rangle)\,\{\llbracket A \rrbracket_{\delta, X \mapsto \Phi}\} \\
\llbracket \exists X.\, A \rrbracket_\delta &\triangleq \lambda v.\, \exists \Phi : \mathsf{SemType}.\, \exists w.\, v = \mathtt{pack}\langle w \rangle \land \llbracket A \rrbracket_{\delta, X \mapsto \Phi}\, w
\end{aligned}$$

Important observations:

- Each type is given a semantic interpretation via the Curry-Howard correspondence
- Instead of Rocq's Prop we can use **a logic with more fancy connectives** to interpret challenging types (*e.g.*, substructural types)

A more feature-rich logical relation

$$[\![_]\!]_\delta : \text{Type} \to \text{SemType} \quad \text{where} \quad \text{SemType} \triangleq \text{Val} \to \text{Prop}$$

$$[\![\mathbf{Z}]\!]_\delta \triangleq \lambda v.\, v \in \mathbb{Z}$$

$$[\![A \times B]\!]_\delta \triangleq \lambda v.\, \exists v_1, v_2.\, v = (v_1, v_2) \wedge [\![A]\!]_\delta\, v_1 \wedge [\![B]\!]_\delta\, v_2$$

$$[\![A \to B]\!]_\delta \triangleq \lambda v.\, \forall w.\, [\![A]\!]_\delta\, w \Rightarrow \text{wp}\,(v\ \ w)\,\{[\![B]\!]_\delta\}$$

$$[\![\forall X.\, A]\!]_\delta \triangleq \lambda v.\, \forall \Phi : \text{SemType}.\, \text{wp}\,(v\langle\rangle)\,\{[\![A]\!]_{\delta, X \mapsto \Phi}\}$$

$$[\![\exists X.\, A]\!]_\delta \triangleq \lambda v.\, \exists \Phi : \text{SemType}.\, \exists w.\, v = \texttt{pack}\langle w\rangle \wedge [\![A]\!]_{\delta, X \mapsto \Phi}\, w$$

Important observations:

- Each type is given a semantic interpretation via the Curry-Howard correspondence
- Instead of Rocq's Prop we can use **a logic with more fancy connectives** to interpret challenging types (*e.g.*, substructural types)

separation logic

A more feature-rich logical relation

$$[\![_]\!]_\delta : \mathsf{Type} \to \mathsf{SemType} \quad \text{where} \quad \mathsf{SemType} \triangleq \mathsf{Val} \to \mathsf{Prop}$$

$$[\![\mathbf{Z}]\!]_\delta \triangleq \lambda v.\, v \in \mathbb{Z}$$

$$[\![A \times B]\!]_\delta \triangleq \lambda v.\, \exists v_1, v_2.\, v = (v_1, v_2) \wedge [\![A]\!]_\delta\, v_1 \wedge [\![B]\!]_\delta\, v_2$$

$$[\![A \to B]\!]_\delta \triangleq \lambda v.\, \forall w.\, [\![A]\!]_\delta\, w \Rightarrow \mathsf{wp}\,(v\ \ w)\,\{[\![B]\!]_\delta\}$$

$$[\![\forall X.\, A]\!]_\delta \triangleq \lambda v.\, \forall \Phi : \mathsf{SemType}.\, \mathsf{wp}\,(v\langle\rangle)\,\{[\![A]\!]_{\delta, X \mapsto \Phi}\}$$

$$[\![\exists X.\, A]\!]_\delta \triangleq \lambda v.\, \exists \Phi : \mathsf{SemType}.\, \exists w.\, v = \mathtt{pack}\langle w\rangle \wedge [\![A]\!]_{\delta, X \mapsto \Phi}\, w$$

Important observations:

- Each type is given a semantic interpretation via the Curry-Howard correspondence
- Instead of Rocq's Prop we can use **a logic with more fancy connectives** to interpret challenging types (*e.g.*, substructural types)

higher-order separation logic

A more feature-rich logical relation

$$[\![_]\!]_\delta : \mathsf{Type} \to \mathsf{SemType} \quad \text{where} \quad \mathsf{SemType} \triangleq \mathsf{Val} \to \mathsf{Prop}$$

$$[\![\mathbf{Z}]\!]_\delta \triangleq \lambda v.\, v \in \mathbb{Z}$$

$$[\![A \times B]\!]_\delta \triangleq \lambda v.\, \exists v_1, v_2.\, v = (v_1, v_2) \land [\![A]\!]_\delta\, v_1 \land [\![B]\!]_\delta\, v_2$$

$$[\![A \to B]\!]_\delta \triangleq \lambda v.\, \forall w.\, [\![A]\!]_\delta\, w \Rightarrow \mathsf{wp}\, (v\ w)\, \{[\![B]\!]_\delta\}$$

$$[\![\forall X.\, A]\!]_\delta \triangleq \lambda v.\, \forall \Phi : \mathsf{SemType}.\, \mathsf{wp}\, (v\langle\rangle)\, \{[\![A]\!]_{\delta, X \mapsto \Phi}\}$$

$$[\![\exists X.\, A]\!]_\delta \triangleq \lambda v.\, \exists \Phi : \mathsf{SemType}.\, \exists w.\, v = \mathtt{pack}\langle w\rangle \land [\![A]\!]_{\delta, X \mapsto \Phi}\, w$$

Important observations:

- Each type is given a semantic interpretation via the Curry-Howard correspondence
- Instead of Rocq's Prop we can use **a logic with more fancy connectives** to interpret challenging types (*e.g.*, substructural types)

higher-order concurrent separation logic

A more feature-rich logical relation

$$[\![_]\!]_\delta : \mathsf{Type} \to \mathsf{SemType} \quad \text{where} \quad \mathsf{SemType} \triangleq \mathsf{Val} \to \mathsf{Prop}$$

$$[\![\mathbf{Z}]\!]_\delta \triangleq \lambda v.\, v \in \mathbb{Z}$$

$$[\![A \times B]\!]_\delta \triangleq \lambda v.\, \exists v_1, v_2.\, v = (v_1, v_2) \wedge [\![A]\!]_\delta\, v_1 \wedge [\![B]\!]_\delta\, v_2$$

$$[\![A \to B]\!]_\delta \triangleq \lambda v.\, \forall w.\, [\![A]\!]_\delta\, w \Rightarrow \mathsf{wp}\,(v\ w)\,\{[\![B]\!]_\delta\}$$

$$[\![\forall X.\, A]\!]_\delta \triangleq \lambda v.\, \forall \Phi : \mathsf{SemType}.\, \mathsf{wp}\,(v\langle\rangle)\,\{[\![A]\!]_{\delta, X \mapsto \Phi}\}$$

$$[\![\exists X.\, A]\!]_\delta \triangleq \lambda v.\, \exists \Phi : \mathsf{SemType}.\, \exists w.\, v = \mathtt{pack}\langle w \rangle \wedge [\![A]\!]_{\delta, X \mapsto \Phi}\, w$$

Important observations:

- Each type is given a semantic interpretation via the Curry-Howard correspondence
- Instead of Rocq's Prop we can use **a logic with more fancy connectives** to interpret challenging types (*e.g.*, substructural types)

Substructural types

Intuition and simple typing rules

Variables can be used **exactly (linear)** or **at-most (affine)** once

For example, $\lambda f.\, \lambda x.\, f\ x\ x$ is **not typeable**

Substructural types

Intuition and simple typing rules

Variables can be used **exactly (linear)** or **at-most (affine)** once

For example, $\lambda f.\, \lambda x.\, f\ x\ x$ is **not typeable**

Useful when types denote ownership of resources

- Session types: Channels – Ensure protocol compliance
- Rust: Memory locations – Avoid use after free and data races

Semantic typing for a substructural type system

Semantic interpretation of types:

$$[\![_]\!] : \mathsf{Type} \to \mathsf{SemType} \quad \text{where} \quad \mathsf{SemType} \triangleq \mathsf{Val} \to \mathsf{sepProp}$$

Semantic typing for a substructural type system

Semantic interpretation of types:

$$[\![_]\!] : \mathsf{Type} \to \mathsf{SemType} \quad \text{where} \quad \mathsf{SemType} \triangleq \mathsf{Val} \to \mathsf{sepProp}$$

$$[\![\mathbf{Z}]\!] \triangleq \lambda v.\, v \in \mathbb{Z}$$

Semantic typing for a substructural type system

Semantic interpretation of types:

$$[\![_]\!] : \mathsf{Type} \to \mathsf{SemType} \quad \text{where} \quad \mathsf{SemType} \triangleq \mathsf{Val} \to \mathsf{sepProp}$$

$$[\![\mathbf{Z}]\!] \triangleq \lambda v.\, v \in \mathbb{Z}$$

$$[\![A \times B]\!] \triangleq \lambda v.\, \exists v_1, v_2.\, v = (v_1, v_2) * [\![A]\!] v_1 * [\![B]\!] v_2$$

Semantic typing for a substructural type system

Semantic interpretation of types:

$$\llbracket _ \rrbracket : \mathsf{Type} \to \mathsf{SemType} \quad \text{where} \quad \mathsf{SemType} \triangleq \mathsf{Val} \to \mathsf{sepProp}$$

$$\llbracket \mathbf{Z} \rrbracket \triangleq \lambda v.\, v \in \mathbb{Z}$$

$$\llbracket A \times B \rrbracket \triangleq \lambda v.\, \exists v_1, v_2.\, v = (v_1, v_2) * \llbracket A \rrbracket v_1 * \llbracket B \rrbracket v_2$$

$$\llbracket A \multimap B \rrbracket \triangleq \lambda v.\, \forall w.\, \llbracket A \rrbracket w \mathrel{-\!\!*} \mathsf{wp}\,(v\ w)\,\{\llbracket B \rrbracket\}$$

Semantic typing for a substructural type system

Semantic interpretation of types:

$$[\![_]\!] : \mathsf{Type} \to \mathsf{SemType} \quad \text{where} \quad \mathsf{SemType} \triangleq \mathsf{Val} \to \mathsf{sepProp}$$

$$
\begin{aligned}
[\![\mathbf{Z}]\!] &\triangleq \lambda v.\, v \in \mathbb{Z} \\
[\![A \times B]\!] &\triangleq \lambda v.\, \exists v_1, v_2.\, v = (v_1, v_2) * [\![A]\!] v_1 * [\![B]\!] v_2 \\
[\![A \multimap B]\!] &\triangleq \lambda v.\, \forall w.\, [\![A]\!] w \mathrel{-\!\!*} \mathsf{wp}\,(v\ w)\,\{[\![B]\!]\} \\
[\![\mathtt{ref}_{\mathsf{uniq}}(A)]\!] &\triangleq \lambda v.\, v \in \mathsf{Loc} * \exists w.\, v \mapsto w * [\![A]\!] w
\end{aligned}
$$

Semantic typing for a substructural type system

Semantic interpretation of types:

$$\llbracket _ \rrbracket : \mathsf{Type} \to \mathsf{SemType} \quad \text{where} \quad \mathsf{SemType} \triangleq \mathsf{Val} \to \mathsf{sepProp}$$

$$\llbracket \mathbf{Z} \rrbracket \triangleq \lambda v.\, v \in \mathbb{Z}$$

$$\llbracket A \times B \rrbracket \triangleq \lambda v.\, \exists v_1, v_2.\, v = (v_1, v_2) * \llbracket A \rrbracket v_1 * \llbracket B \rrbracket v_2$$

$$\llbracket A \multimap B \rrbracket \triangleq \lambda v.\, \forall w.\, \llbracket A \rrbracket w \mathrel{-\!\!*} \mathsf{wp}\,(v\ w)\,\{\llbracket B \rrbracket\}$$

$$\llbracket \mathtt{ref}_{\mathsf{uniq}}(A) \rrbracket \triangleq \lambda v.\, v \in \mathsf{Loc} * \exists w.\, v \mapsto w * \llbracket A \rrbracket w$$

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

Semantic typing for a substructural type system

Semantic interpretation of types:

$$[\![_]\!] : \mathsf{Type} \to \mathsf{SemType} \quad \text{where} \quad \mathsf{SemType} \triangleq \mathsf{Val} \to \mathsf{iProp}$$

$$\begin{aligned}
[\![\mathbf{Z}]\!] &\triangleq \lambda v.\, v \in \mathbb{Z} \\
[\![A \times B]\!] &\triangleq \lambda v.\, \exists v_1, v_2.\, v = (v_1, v_2) * [\![A]\!] v_1 * [\![B]\!] v_2 \\
[\![A \multimap B]\!] &\triangleq \lambda v.\, \forall w.\, [\![A]\!] w \mathrel{-\!\!*} \mathsf{wp}\,(v\ w)\,\{[\![B]\!]\} \\
[\![\mathtt{ref}_{\mathsf{uniq}}(A)]\!] &\triangleq \lambda v.\, v \in \mathsf{Loc} * \exists w.\, v \mapsto w * [\![A]\!] w \\
[\![\mathtt{ref}_{\mathsf{shr}}(A)]\!] &\triangleq \lambda v.\, v \in \mathsf{Loc} * \boxed{\exists w.\, v \mapsto w * [\![A]\!] w}
\end{aligned}$$

Iris **invariant** $\boxed{P}$ $\approx$ knowledge that P holds at all times (invariantly)

Semantic typing for a substructural type system

Semantic interpretation of types:

$$\llbracket _ \rrbracket : \mathsf{Type} \to \mathsf{SemType} \quad \text{where} \quad \mathsf{SemType} \triangleq \mathsf{Val} \to \mathsf{iProp}$$

$$\begin{aligned}
\llbracket \mathbf{Z} \rrbracket &\triangleq \lambda v.\, v \in \mathbb{Z} \\
\llbracket A \times B \rrbracket &\triangleq \lambda v.\, \exists v_1, v_2.\, v = (v_1, v_2) * \llbracket A \rrbracket v_1 * \llbracket B \rrbracket v_2 \\
\llbracket A \multimap B \rrbracket &\triangleq \lambda v.\, \forall w.\, \llbracket A \rrbracket w \mathrel{-\!\!*} \mathsf{wp}\,(v\ w)\,\{\llbracket B \rrbracket\} \\
\llbracket \mathtt{ref}_{\mathsf{uniq}}(A) \rrbracket &\triangleq \lambda v.\, v \in \mathsf{Loc} * \exists w.\, v \mapsto w * \llbracket A \rrbracket w \\
\llbracket \mathtt{ref}_{\mathsf{shr}}(A) \rrbracket &\triangleq \lambda v.\, v \in \mathsf{Loc} * \boxed{\exists w.\, v \mapsto w * \llbracket A \rrbracket w}
\end{aligned}$$

This scales—pick the right Iris features to interpret your favorite types

Summary: Recipe for verifying a type system in Iris

1. Define the syntax and operational semantics for your language
2. Build a program logic using Iris, *i.e.*, define WP, $\mapsto$, *etc.*
3. Verify separation logic APIs for your "unsafe" libraries
4. Define a logical relation and semantic typing judgment
5. Prove semantic typing rules/fundamental theorem
6. Profit

Summary: Recipe for verifying a type system in Iris

1. Define the syntax and operational semantics for your language
   Decide what operations should be primitives or implemented as “unsafe”
2. Build a program logic using Iris, *i.e.*, define WP, $\mapsto$, *etc.*
3. Verify separation logic APIs for your “unsafe” libraries
4. Define a logical relation and semantic typing judgment
5. Prove semantic typing rules/fundamental theorem
6. Profit

Summary: Recipe for verifying a type system in Iris

1. Define the syntax and operational semantics for your language
   Decide what operations should be primitives or implemented as "unsafe"
2. Build a program logic using Iris, *i.e.*, define WP, $\mapsto$, *etc.*
   Iris provides reusable building blocks for defining and verifying program logics
3. Verify separation logic APIs for your "unsafe" libraries
4. Define a logical relation and semantic typing judgment
5. Prove semantic typing rules/fundamental theorem
6. Profit

Summary: Recipe for verifying a type system in Iris

1. Define the syntax and operational semantics for your language
   Decide what operations should be primitives or implemented as "unsafe"
2. Build a program logic using Iris, *i.e.*, define WP, $\mapsto$, *etc.*
   Iris provides reusable building blocks for defining and verifying program logics
3. Verify separation logic APIs for your "unsafe" libraries
   Make use of invariants and ghost state provided by Iris
4. Define a logical relation and semantic typing judgment
5. Prove semantic typing rules/fundamental theorem
6. Profit

Summary: Recipe for verifying a type system in Iris

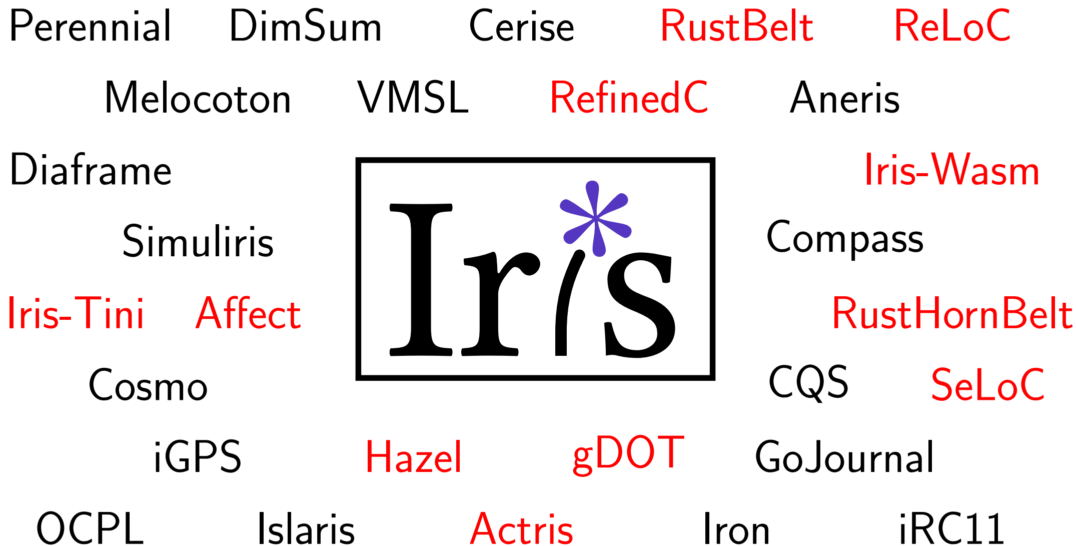
1. Define the syntax and operational semantics for your language
   Decide what operations should be primitives or implemented as "unsafe"
2. Build a program logic using Iris, *i.e.*, define WP, $\mapsto$, *etc.*
   Iris provides reusable building blocks for defining and verifying program logics
3. Verify separation logic APIs for your "unsafe" libraries
   Make use of invariants and ghost state provided by Iris
4. Define a logical relation and semantic typing judgment
   Interpret type formers using suitable logical connectives through Curry-Howard
5. Prove semantic typing rules/fundamental theorem

6. Profit

Summary: Recipe for verifying a type system in Iris

1. Define the syntax and operational semantics for your language
   Decide what operations should be primitives or implemented as "unsafe"
2. Build a program logic using Iris, *i.e.*, define WP, $\mapsto$, *etc.*
   Iris provides reusable building blocks for defining and verifying program logics
3. Verify separation logic APIs for your "unsafe" libraries
   Make use of invariants and ghost state provided by Iris
4. Define a logical relation and semantic typing judgment
   Interpret type formers using suitable logical connectives through Curry-Howard
5. Prove semantic typing rules/fundamental theorem
   Most of the heavy lifting is done by the Hoare/WP rules in Iris
6. Profit

The logical approach in Iris scales



Future work: Going beyond safety

- Applying the logical approach to deadlock freedom, resource leak freedom, liveness, non-interference remains challenging
- Different models of concurrent separation logic/Iris need to be explored: linear (instead of affine), transfinite, *etc.*
- We have initial versions for specific languages
- But we do not have the right Iris-style abstractions to build these logics modularly
- Nor to easily combine different PL features in one type safety proof