Proving Type Safety Using Separation Logic

Robbert Krebbers

Radboud University Nijmegen, The Netherlands

January 23, 2026 @ CYCLIC winter school, Enschede, NL

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 $\mathrm{safe}(e)$

The old problem of proving "type safety":

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

Goal of this lecture:

- 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 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 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 safe(e)

2. **Fundamental theorem:** If $\vdash e : A$ implies $\vDash e : A$

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 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 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 $\vdash e : A$

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

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 with channels implemented as an “unsafe” library

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"):

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

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

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

Semantic typing for STLC

Semantic interpretation of types ("logical relation"):

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

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

Semantic typing for STLC

Semantic interpretation of types ("logical relation"):

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

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

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

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

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:

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

Proofs of key properties

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

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 $\vDash e : A$

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 $\mathsf{safe}(e)$
   Holds by definition $\vDash e : A \;=\; \mathsf{wp}\; e\,\{\llbracket A \rrbracket\} \;=\; \mathsf{safe}(e) \wedge \ldots$
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 ($\vdash$)

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

Proof of the fundamental theorem

Reasoning about the operational semantics is encapsulated by the WP rules

$$\textsc{wp-val}\quad \frac{\Phi\ v}{\mathsf{wp}\ v\ \{\Phi\}} \qquad\qquad \textsc{wp-bind}\quad \frac{\mathsf{wp}\ e\ \{\Psi\} \qquad (\forall v.\ \Psi\ v \Rightarrow \mathsf{wp}\ K[\,v\,]\ \{\Phi\})}{\mathsf{wp}\ K[\,e\,]\ \{\Phi\}}$$

Proof of the fundamental theorem

Reasoning about the operational semantics is encapsulated by the WP rules

$$\frac{\Phi\ v}{\mathsf{wp}\ v\ \{\Phi\}}\ \textsc{wp-val} \qquad \frac{\mathsf{wp}\ e\ \{\Psi\} \qquad (\forall v.\ \Psi\ v \Rightarrow \mathsf{wp}\ K[\,v\,]\ \{\Phi\})}{\mathsf{wp}\ K[\,e\,]\ \{\Phi\}}\ \textsc{wp-bind}$$

Example: Proof of the semantic typing rule for application

$$\frac{}{\vDash e_1\ e_2 : B}$$

Proof of the fundamental theorem

Reasoning about the operational semantics is encapsulated by the WP rules

$$\textsc{wp-val}\ \frac{\Phi\ v}{\mathsf{wp}\ v\ \{\Phi\}} \qquad \textsc{wp-bind}\ \frac{\mathsf{wp}\ e\ \{\Psi\} \qquad (\forall v.\ \Psi\ v \Rightarrow \mathsf{wp}\ K[\,v\,]\ \{\Phi\})}{\mathsf{wp}\ K[\,e\,]\ \{\Phi\}}$$

Example: Proof of the semantic typing rule for application

$$\frac{\dfrac{}{\mathsf{wp}\ (e_1\ \ e_2)\ \{[\![B]\!]\}}}{\vDash e_1\ \ e_2 : B}$$

Proof of the fundamental theorem

Reasoning about the operational semantics is encapsulated by the WP rules

$$\textsc{wp-val}\ \frac{\Phi\ v}{\mathsf{wp}\ v\ \{\Phi\}} \qquad\qquad \textsc{wp-bind}\ \frac{\mathsf{wp}\ e\ \{\Psi\} \qquad (\forall v.\, \Psi\ v \Rightarrow \mathsf{wp}\ K[\,v\,]\ \{\Phi\})}{\mathsf{wp}\ K[\,e\,]\ \{\Phi\}}$$

Example: Proof of the semantic typing rule for application

$$\dfrac{\dfrac{\dfrac{}{\mathsf{wp}\ e_2\ \{[\![A]\!]\}} \qquad \dfrac{}{[\![A]\!] v_2 \Rightarrow \mathsf{wp}\ (e_1\ \ v_2)\ \{[\![B]\!]\}}}{\mathsf{wp}\ (e_1\ \ e_2)\ \{[\![B]\!]\}}}{\vDash e_1\ \ e_2 : B}$$

Proof of the fundamental theorem

Reasoning about the operational semantics is encapsulated by the WP rules

$$\textsc{wp-val}\ \frac{\Phi\ v}{\mathsf{wp}\ v\ \{\Phi\}} \qquad \textsc{wp-bind}\ \frac{\mathsf{wp}\ e\ \{\Psi\} \qquad (\forall v.\, \Psi\ v \Rightarrow \mathsf{wp}\ K[\,v\,]\ \{\Phi\})}{\mathsf{wp}\ K[\,e\,]\ \{\Phi\}}$$

Example: Proof of the semantic typing rule for application

$$\frac{\dfrac{\dfrac{\vDash e_2 : A}{\mathsf{wp}\ e_2\ \{\llbracket A \rrbracket\}} \qquad \dfrac{}{\llbracket A \rrbracket v_2 \Rightarrow \mathsf{wp}\ (e_1\ \ v_2)\ \{\llbracket B \rrbracket\}}}{\mathsf{wp}\ (e_1\ \ e_2)\ \{\llbracket B \rrbracket\}}}{\vDash e_1\ \ e_2 : B}$$

Proof of the fundamental theorem

Reasoning about the operational semantics is encapsulated by the WP rules

$$\textsc{wp-val}\ \frac{\Phi\ v}{\mathsf{wp}\ v\ \{\Phi\}} \qquad \textsc{wp-bind}\ \frac{\mathsf{wp}\ e\ \{\Psi\} \qquad (\forall v.\, \Psi\ v \Rightarrow \mathsf{wp}\ K[\,v\,]\ \{\Phi\})}{\mathsf{wp}\ K[\,e\,]\ \{\Phi\}}$$

Example: Proof of the semantic typing rule for application

$$\dfrac{\dfrac{\dfrac{\vDash e_2 : A}{\mathsf{wp}\ e_2\ \{[\![A]\!]\}} \qquad \dfrac{\overline{\mathsf{wp}\ e_1\ \{[\![A \to B]\!]\}} \qquad \overline{[\![A \to B]\!] v_1 \Rightarrow [\![A]\!] v_2 \Rightarrow \mathsf{wp}\ (v_1\ v_2)\ \{[\![B]\!]\}}}{[\![A]\!] v_2 \Rightarrow \mathsf{wp}\ (e_1\ v_2)\ \{[\![B]\!]\}}}{\mathsf{wp}\ (e_1\ e_2)\ \{[\![B]\!]\}}}{\vDash e_1\ e_2 : B}$$

Proof of the fundamental theorem

Reasoning about the operational semantics is encapsulated by the WP rules

$$\textsc{wp-val}\ \frac{\Phi\ v}{\mathsf{wp}\ v\ \{\Phi\}} \qquad\qquad \textsc{wp-bind}\ \frac{\mathsf{wp}\ e\ \{\Psi\} \qquad (\forall v.\, \Psi\ v \Rightarrow \mathsf{wp}\ K[\,v\,]\ \{\Phi\})}{\mathsf{wp}\ K[\,e\,]\ \{\Phi\}}$$

Example: Proof of the semantic typing rule for application

$$\dfrac{\dfrac{\vDash e_2 : A}{\mathsf{wp}\ e_2\ \{[\![A]\!]\}} \qquad \dfrac{\dfrac{\vDash e_1 : A \to B}{\mathsf{wp}\ e_1\ \{[\![A \to B]\!]\}} \qquad \dfrac{}{[\![A \to B]\!] v_1 \Rightarrow [\![A]\!] v_2 \Rightarrow \mathsf{wp}\ (v_1\ \ v_2)\ \{[\![B]\!]\}}}{[\![A]\!] v_2 \Rightarrow \mathsf{wp}\ (e_1\ \ v_2)\ \{[\![B]\!]\}}}{\dfrac{\mathsf{wp}\ (e_1\ \ e_2)\ \{[\![B]\!]\}}{\vDash e_1\ \ e_2 : B}}$$

Proof of the fundamental theorem

Reasoning about the operational semantics is encapsulated by the WP rules

$$\dfrac{\Phi\ v}{\mathsf{wp}\ v\ \{\Phi\}}\ \textsc{wp-val} \qquad \dfrac{\mathsf{wp}\ e\ \{\Psi\} \qquad (\forall v.\, \Psi\ v \Rightarrow \mathsf{wp}\ K[\,v\,]\ \{\Phi\})}{\mathsf{wp}\ K[\,e\,]\ \{\Phi\}}\ \textsc{wp-bind}$$

Example: Proof of the semantic typing rule for application

$$\dfrac{\dfrac{\dfrac{\vDash e_2 : A}{\mathsf{wp}\ e_2\ \{[\![A]\!]\}} \qquad \dfrac{\dfrac{\vDash e_1 : A \to B}{\mathsf{wp}\ e_1\ \{[\![A \to B]\!]\}} \qquad \dfrac{}{[\![A \to B]\!] v_1 \Rightarrow [\![A]\!] v_2 \Rightarrow \mathsf{wp}\ (v_1\ v_2)\ \{[\![B]\!]\}}}{[\![A]\!] v_2 \Rightarrow \mathsf{wp}\ (e_1\ v_2)\ \{[\![B]\!]\}}}{\mathsf{wp}\ (e_1\ e_2)\ \{[\![B]\!]\}}}{\vDash e_1\ e_2 : B}$$

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

An "unsafe" fixpoint combinator

Consider a strict version of Curry's fixpoint operator:

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

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?

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

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

Now let us add polymorphism

Polymorphism and existential types (System F)

Typing rules

T-TLAM
$$\frac{\Gamma \vdash e : A}{\Gamma \vdash \Lambda X.\, e : \forall X.\, A}$$

T-TAPP
$$\frac{\Gamma \vdash e : \forall X.\, A}{\Gamma \vdash e\langle B \rangle : A[B/X]}$$

T-PACK
$$\frac{\Gamma \vdash e : A[B/X]}{\Gamma \vdash \mathtt{pack}\, \langle B, e \rangle : \exists X.\, A}$$

T-MATCH-EX
$$\frac{\Gamma \vdash e : \exists X.\, A \qquad \Gamma, x : A \vdash e_2 : B}{\Gamma \vdash \mathtt{match}\; e\; \mathtt{with}\; \mathtt{pack}\, \langle X, x \rangle \Rightarrow e_2\; \mathtt{end} : B}$$

Polymorphism and existential types (System F)

Typing rules

$$\textsc{T-tlam}\ \frac{\Gamma \vdash e : A}{\Gamma \vdash \Lambda X.\, e : \forall X.\, A}$$

$$\textsc{T-tapp}\ \frac{\Gamma \vdash e : \forall X.\, A}{\Gamma \vdash e\langle B \rangle : A[B/X]}$$

$$\textsc{T-pack}\ \frac{\Gamma \vdash e : A[B/X]}{\Gamma \vdash \mathtt{pack}\ \langle B, e\rangle : \exists X.\, A}$$

$$\textsc{T-match-ex}\ \frac{\Gamma \vdash e : \exists X.\, A \qquad \Gamma, x : A \vdash e_2 : B}{\Gamma \vdash \mathtt{match}\ e\ \mathtt{with}\ \mathtt{pack}\ \langle X, x\rangle \Rightarrow e_2\ \mathtt{end} : B}$$

For safety, the type annotations are irrelevant, so we erase them

Polymorphism and existential types (System F)

Typing rules

$$\textsc{T-tlam}\ \frac{\Gamma \vdash e : A}{\Gamma \vdash \Lambda.\, e : \forall X.\, A} \qquad \textsc{T-tapp}\ \frac{\Gamma \vdash e : \forall X.\, A}{\Gamma \vdash e\langle\rangle : A[B/X]}$$

$$\textsc{T-pack}\ \frac{\Gamma \vdash e : A[B/X]}{\Gamma \vdash \texttt{pack}\langle e\rangle : \exists X.\, A} \qquad \textsc{T-match-ex}\ \frac{\Gamma \vdash e : \exists X.\, A \qquad \Gamma, x : A \vdash e_2 : B}{\Gamma \vdash \texttt{match}\ e\ \texttt{with}\ \texttt{pack}\langle x\rangle \Rightarrow e_2\ \texttt{end} : B}$$

For safety, the type annotations are irrelevant, so we erase them

Polymorphism and existential types (System F)

Naive attempt at extending the logical relation

$$\llbracket \forall X.\, A \rrbracket \triangleq \lambda v.\, \forall (B : \mathsf{Type}).\, \mathsf{wp}\ (v\langle\rangle)\ \{\llbracket A[B/X] \rrbracket\}$$

$$\llbracket \exists X.\, A \rrbracket \triangleq \lambda v.\, \exists (B : \mathsf{Type}).\, \exists w.\, (v = \mathtt{pack}\langle w \rangle) * \llbracket A[B/X] \rrbracket w$$

Polymorphism and existential types (System F)

Naive attempt at extending the logical relation

$$\llbracket \forall X.\, A \rrbracket \triangleq \lambda v.\, \forall (B : \mathsf{Type}).\, \mathsf{wp}\,(v\langle\rangle)\,\{\llbracket A[B/X] \rrbracket\}$$

$$\llbracket \exists X.\, A \rrbracket \triangleq \lambda v.\, \exists (B : \mathsf{Type}).\, \exists w.\, (v = \mathtt{pack}\langle w \rangle) * \llbracket A[B/X] \rrbracket w$$

Problem: The recursive calls are not well-founded

Polymorphism and existential types (System F)

Correct attempt at extending the logical relation

Inspired by reducibility candidates (Girard) and parametricity (Reynolds):

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

$$\text{and} \quad \delta : \mathsf{Tvar} \xrightarrow{\text{fin}} \mathsf{SemType}$$

$$[\![X]\!]_\delta \triangleq \delta(X)$$

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

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

Polymorphism and existential types (System F)

Correct attempt at extending the logical relation

Inspired by reducibility candidates (Girard) and parametricity (Reynolds):

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

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

Key idea: Quantify over semantic types

Polymorphism and existential types (System F)

Correct attempt at extending the logical relation

Inspired by reducibility candidates (Girard) and parametricity (Reynolds):

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

$$\text{and} \quad \delta : \mathsf{Tvar} \xrightarrow{\text{fin}} \mathsf{SemType}$$

$$\begin{aligned}
[\![X]\!]_\delta &\triangleq \delta(X) \\
[\![\forall X.\, A]\!]_\delta &\triangleq \lambda v.\, \forall \Phi : \mathsf{SemType}.\, \mathsf{wp}\ (v\langle\rangle)\ \{[\![A]\!]_{\delta, X \mapsto \Phi}\} \\
[\![\exists X.\, A]\!]_\delta &\triangleq \lambda v.\, \exists \Phi : \mathsf{SemType}.\, \exists w.\, v = \mathtt{pack}\langle w\rangle \wedge [\![A]\!]_{\delta, X \mapsto \Phi}\, w
\end{aligned}$$

Key idea: Quantify over semantic types

Fundamentally relies on Rocq's support for higher-order impredicative quantification

Now that we have our baseline version, let us scale it up

Towards "logical typing"

Recall the semantic interpretation of types ("logical relation"):

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

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

Towards "logical typing"

Recall the semantic interpretation of types ("logical relation"):

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

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

Important observations:

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

Towards "logical typing"

Recall the semantic interpretation of types ("logical relation"):

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

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

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)

Towards "logical typing"

Recall the semantic interpretation of types ("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
- Instead of Rocq's Prop we can use **a logic with more fancy connectives** to interpret challenging types (*e.g.*, substructural types)

separation logic

Towards "logical typing"

Recall the semantic interpretation of types ("logical relation"):

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

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

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)

higher-order separation logic

Towards "logical typing"

Recall the semantic interpretation of types ("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
- 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

Towards "logical typing"

Recall the semantic interpretation of types ("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
- 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

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

Affine typing rules:

$$\textsc{T-var}\ \frac{x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \textsc{T-lam}\ \frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x.\, e : A \multimap B} \qquad \textsc{T-app}\ \frac{\Gamma_1 \vdash e_1 : A \multimap B \qquad \Gamma_2 \vdash e_2 : A}{\Gamma_1 \uplus \Gamma_2 \vdash e_1\ e_2 : B}$$

split the context to ensure at-most-once usage

Key thing to remember: Separation logic is a perfect fit for logical relations for substructural type systems

Separation logic [O'Hearn, Reynolds, Yang; CSL'01]

Propositions P, Q denote ownership of resources

Separating conjunction $P * Q$:
The resources consists of separate parts satisfying P and Q

Basic example:

$$\{\ell_1 \mapsto v_1 * \ell_2 \mapsto v_2\}\mathit{swap}\ \ell_1\ \ell_2\{\ell_1 \mapsto v_2 * \ell_2 \mapsto v_1\}$$

the $*$ ensures that ℓ_1 and ℓ_2 are different memory locations

The simple heap model of separation logic

The semantic domains:

$$\ell \in \mathsf{Loc} \triangleq \mathbb{N}$$

$$\sigma \in \mathsf{Heap} \triangleq \mathsf{Loc} \xrightarrow{\text{fin}} \mathsf{Val}$$

$$P, Q \in \mathsf{sepProp} \triangleq \mathsf{Heap} \to \mathsf{Prop}$$

The simple heap model of separation logic

The semantic domains:

$$\ell \in \mathsf{Loc} \triangleq \mathbb{N}$$

$$\sigma \in \mathsf{Heap} \triangleq \mathsf{Loc} \xrightharpoonup{\text{fin}} \mathsf{Val}$$

$$P, Q \in \mathsf{sepProp} \triangleq \mathsf{Heap} \to \mathsf{Prop}$$

Entailment:

$$P \vdash Q \triangleq \forall \sigma.\, P\sigma \to Q\sigma$$

The simple heap model of separation logic

The semantic domains:

$$\ell \in \mathsf{Loc} \triangleq \mathbb{N}$$

$$\sigma \in \mathsf{Heap} \triangleq \mathsf{Loc} \xrightarrow{\text{fin}} \mathsf{Val}$$

$$P, Q \in \mathsf{sepProp} \triangleq \mathsf{Heap} \to \mathsf{Prop}$$

Entailment:

$$P \vdash Q \triangleq \forall \sigma.\, P\sigma \to Q\sigma$$

The connectives of separation logic:

$$\ell \mapsto v \triangleq \lambda \sigma.\, \sigma(\ell) = v$$

$$P \wedge Q \triangleq \lambda \sigma.\, P\sigma \wedge Q\sigma$$

$$P * Q \triangleq \lambda \sigma.\, \exists \sigma_1, \sigma_2.\, \sigma = \sigma_1 \uplus \sigma_2 \wedge P\sigma_1 \wedge Q\sigma_2$$

$$(\exists x : A.\, P) \triangleq \lambda \sigma.\, \exists x : A.\, P\sigma$$

disjointness of heaps, hidden by $*$

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:

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

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

Semantic typing for a substructural type system

Semantic interpretation of types:

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

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

Weakest precondition of separation logic:

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

$$\begin{aligned}
\mathsf{wp}\ e\,\{\Phi\} \triangleq \lambda \sigma.\, \forall \sigma_f.\, &\mathsf{safe}(\sigma \uplus \sigma_f, e) \wedge \\
&(\forall v, \sigma'.\, (\sigma \uplus \sigma_f, e) \to^* (\sigma', v) \Rightarrow \\
&\qquad \exists \sigma''.\, \sigma' = \sigma'' \uplus \sigma_f \wedge \Phi\ v\ \sigma'')
\end{aligned}$$

Semantic typing for a substructural type system

Semantic interpretation of types:

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

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

Semantic typing for a substructural type system

Semantic interpretation of types:

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

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

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

Interlude: Weakest preconditions versus Hoare triples

In Iris, Hoare triples are not primitive, but encoded in terms of weakest preconditions:

- Weakest preconditions work nicer in Rocq
- Weakest preconditions are a better fit for defining logical relations

Interlude: Weakest preconditions versus Hoare triples

In Iris, Hoare triples are not primitive, but encoded in terms of weakest preconditions:

- Weakest preconditions work nicer in Rocq
- Weakest preconditions are a better fit for defining logical relations

$$\{P\}\; e\; \{\Phi\} \triangleq P \vdash \mathsf{wp}\; e\; \{\Phi\}$$

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

$$\begin{aligned}\mathsf{wp}\; e\; \{\Phi\} \triangleq \lambda\sigma.\, \forall \sigma_f.\, & \mathsf{safe}(\sigma \uplus \sigma_f, e) \wedge \\ & (\forall v, \sigma'.\, (\sigma \uplus \sigma_f, e) \to^* (\sigma', v) \Rightarrow \\ & \qquad \exists \sigma''.\, \sigma' = \sigma'' \uplus \sigma_f \wedge \Phi\; v\; \sigma'')\end{aligned}$$

$$(\vdash) : \mathsf{sepProp} \to \mathsf{sepProp} \to \mathsf{Prop}$$

$$P \vdash Q \triangleq \forall \sigma.\, P\sigma \to Q\sigma$$

Now let us add recursive types

Iso-recursive types

Typing rules

$$\textsc{T-fold}\ \frac{\Gamma \vdash e : A[\mu X.\, A/X]}{\Gamma \vdash \texttt{fold}\, e : \mu X.\, A}$$

$$\textsc{T-unfold}\ \frac{\Gamma \vdash e : \mu X.\, A}{\Gamma \vdash \texttt{unfold}\, e : A[\mu X.\, A/X]}$$

Iso-recursive types

Typing rules

$$\textsc{T-fold}\quad \frac{\Gamma \vdash e : A[\mu X.\, A/X]}{\Gamma \vdash \texttt{fold}\, e : \mu X.\, A}$$

$$\textsc{T-unfold}\quad \frac{\Gamma \vdash e : \mu X.\, A}{\Gamma \vdash \texttt{unfold}\, e : A[\mu X.\, A/X]}$$

For example, $\texttt{linkedlist}\, B \triangleq \mu X.\, \texttt{ref}_{\mathsf{uniq}}(() + (B \times X))$

$$\begin{tikzcd}
\texttt{linkedlist}\, B \arrow[r, bend left, "\texttt{unfold}"] & \texttt{ref}_{\mathsf{uniq}}(() + (B \times \texttt{linkedlist}\, B)) \arrow[l, bend left, "\texttt{fold}"]
\end{tikzcd}$$

Iso-recursive types

Logical relation

$$\llbracket \_ \rrbracket_\delta : \mathsf{Type} \to \mathsf{SemType} \quad \text{where} \quad \mathsf{SemType} \triangleq \mathsf{Val} \to \mathsf{iProp}$$
$$\text{and} \quad \delta : \mathsf{Tvar} \xrightarrow{\text{fin}} \mathsf{SemType}$$

$$\llbracket X \rrbracket_\delta \triangleq \delta(X)$$
$$\llbracket \mu X.\, A \rrbracket_\delta \triangleq \lambda v.\, \exists w.\, (v = \texttt{fold}\, w) * \quad \llbracket A \rrbracket_{\delta, X \mapsto \llbracket \mu X.\, A \rrbracket_\delta}\, w$$

Iso-recursive types

Logical relation

$$\llbracket \_ \rrbracket_\delta : \mathsf{Type} \to \mathsf{SemType} \quad \text{where} \quad \mathsf{SemType} \triangleq \mathsf{Val} \to \mathsf{iProp}$$
$$\text{and} \quad \delta : \mathsf{Tvar} \xrightarrow{\text{fin}} \mathsf{SemType}$$

$$\llbracket X \rrbracket_\delta \triangleq \delta(X)$$
$$\llbracket \mu X.\, A \rrbracket_\delta \triangleq \lambda v.\, \exists w.\, (v = \texttt{fold}\, w) * \quad \llbracket A \rrbracket_{\delta, X \mapsto \llbracket \mu X.\, A \rrbracket_\delta}\, w$$

not structurally recursive

Iso-recursive types

Logical relation

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

$$\text{and} \quad \delta : \mathsf{Tvar} \xrightarrow{\text{fin}} \mathsf{SemType}$$

$$[\![X]\!]_\delta \triangleq \delta(X)$$

$$[\![\mu X.\, A]\!]_\delta \triangleq \lambda v.\, \exists w.\, (v = \texttt{fold}\, w) * \triangleright [\![A]\!]_{\delta, X \mapsto [\![\mu X.\, A]\!]_\delta}\, w$$

not structurally recursive

Iris's **later modality** to guard the recursion

The later modality

$$P \vdash \triangleright P \qquad \frac{P \vdash Q}{\triangleright P \vdash \triangleright Q} \qquad (\triangleright P \Rightarrow P) \vdash P$$

Now let us add an “unsafe” library

Typing "unsafe" code: One-shot channels

We can **implement** one-shot channels instead of adding them as primitives to our language (akin to using `unsafe` in Rust):

$$
\begin{aligned}
\mathtt{new}\,() &\triangleq \mathtt{let}\; c = \mathtt{ref}\;\mathtt{None}\;\mathtt{in}\;(c, c) \\
\mathtt{send}\,(c, v) &\triangleq c := \mathtt{Some}\; v \\
\mathtt{recv}\; c &\triangleq \mathtt{let}\; x = {!}\,c\;\mathtt{in} \\
&\quad\; \mathtt{match}\; x\; \mathtt{with} \\
&\quad\;\;\; \mathtt{None} \;\;\;\; \Rightarrow \mathtt{recv}\; c \\
&\quad\; \mid \mathtt{Some}\; v \Rightarrow \mathtt{free}\; c;\;\; v \\
&\quad\; \mathtt{end}
\end{aligned}
$$

Typing "unsafe" code: One-shot channels

We can **implement** one-shot channels instead of adding them as primitives to our language (akin to using `unsafe` in Rust):

$$
\begin{aligned}
\mathtt{new}\,() &\triangleq \mathtt{let}\; c = \mathtt{ref}\;\mathtt{None}\;\mathtt{in}\;(c, c) \\
\mathtt{send}\,(c, v) &\triangleq c := \mathtt{Some}\; v \\
\mathtt{recv}\; c &\triangleq \mathtt{let}\; x = {!}\, c\;\mathtt{in} \\
&\quad\;\; \mathtt{match}\; x\; \mathtt{with} \\
&\quad\;\;\;\; \mathtt{None} \quad\;\, \Rightarrow \mathtt{recv}\; c \\
&\quad\;\; \mid \mathtt{Some}\; v \Rightarrow \mathtt{free}\; c;\;\; v \\
&\quad\;\; \mathtt{end}
\end{aligned}
$$

What would be good typed API for one-shot channels?

Typing "unsafe" code: One-shot channels

We can **implement** one-shot channels instead of adding them as primitives to our language (akin to using `unsafe` in Rust):

$$\begin{aligned}
\mathtt{new}\,() &\triangleq \mathtt{let}\; c = \mathtt{ref\ None\ in}\,(c, c)\\
\mathtt{send}\,(c, v) &\triangleq c := \mathtt{Some}\; v\\
\mathtt{recv}\; c &\triangleq \mathtt{let}\; x = {!}\,c\; \mathtt{in}\\
&\quad\; \mathtt{match}\; x\; \mathtt{with}\\
&\quad\;\; \mathtt{None} \quad\;\; \Rightarrow \mathtt{recv}\; c\\
&\quad\; \mid \mathtt{Some}\; v \Rightarrow \mathtt{free}\; c;\;\; v\\
&\quad\; \mathtt{end}
\end{aligned}$$

What would be good typed API for one-shot channels?

$$\vDash \mathtt{new} : () \multimap {!}A \times {?}A \qquad \vDash \mathtt{send} : {!}A \times A \multimap () \qquad \vDash \mathtt{recv} : {?}A \multimap A$$

Typing "unsafe" code: One-shot channels

We can **implement** one-shot channels instead of adding them as primitives to our language (akin to using `unsafe` in Rust):

$$
\begin{aligned}
\texttt{new}\,() &\triangleq \texttt{let}\; c = \texttt{ref None in}\,(c, c) \\
\texttt{send}\,(c, v) &\triangleq c := \texttt{Some}\; v \\
\texttt{recv}\; c &\triangleq \texttt{let}\; x = !\, c\; \texttt{in} \\
&\quad\; \texttt{match}\; x\; \texttt{with} \\
&\quad\;\;\; \texttt{None} \;\;\;\Rightarrow \texttt{recv}\; c \\
&\quad\; \mid \texttt{Some}\; v \Rightarrow \texttt{free}\; c;\;\; v \\
&\quad\; \texttt{end}
\end{aligned}
$$

What would be good typed API for one-shot channels?

$$
\vDash \texttt{new} : () \multimap\; !A \times ?A \qquad \vDash \texttt{send} : !A \times A \multimap () \qquad \vDash \texttt{recv} : ?A \multimap A
$$

Substructural types are essential: calling `recv` twice causes use-after-free

Typing "unsafe" code: One-shot channels

We can **implement** one-shot channels instead of adding them as primitives to our language (akin to using `unsafe` in Rust):

$$\mathtt{new}\,() \triangleq \mathtt{let}\; c = \mathtt{ref\; None\; in}\,(c, c)$$

$$\mathtt{send}\,(c, v) \triangleq c := \mathtt{Some}\; v$$

One-shot channels + recursive types allow one to embed the whole of higher-order binary session types [Jacobs, ECOOP'22]

$$\mid \mathtt{Some}\; v \Rightarrow \mathtt{free}\; c;\;\; v$$

$$\mathtt{end}$$

What would be good typed API for one-shot channels?

$$\vDash \mathtt{new} : () \multimap\, !A \times ?A \qquad \vDash \mathtt{send} : !A \times A \multimap () \qquad \vDash \mathtt{recv} : ?A \multimap A$$

Substructural types are essential: calling `recv` twice causes use-after-free

Typing “unsafe” code: Recipe

1. Provide a separation logic API for the unsafe operations
   Used to give a logical interpretation $[\![\_]\!]$ of the typed API
2. Prove Hoare style specifications for the unsafe operations
   Used to prove the semantic typing rules

Separation logic API for one-shot channels

Recall the desired typing rules:

$$\vDash \texttt{new}\,() : () \multimap\, !A \times ?A$$
$$\vDash \texttt{send} : !A \times A \multimap ()$$
$$\vDash \texttt{recv} : ?A \multimap A$$

The separation logic API:

$$\{\mathsf{True}\}\ \texttt{new}\,()\ \{(c_1, c_2).\, \mathsf{IsChan}(c_1, \mathsf{Send}, \Phi) * \mathsf{IsChan}(c_2, \mathsf{Recv}, \Phi)\}$$
$$\{\mathsf{IsChan}(c, \mathsf{Send}, \Phi) * \Phi\ v\}\ \texttt{send}\,(c, v)\ \{\mathsf{True}\}$$
$$\{\mathsf{IsChan}(c, \mathsf{Recv}, \Phi)\}\ \texttt{recv}\,c\ \{w.\, \Phi\ w\}$$

Logical typing for channels

Semantic interpretation of types ("logical relation"):

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

$$\llbracket !A \rrbracket \triangleq \lambda c.\, \mathsf{IsChan}(c, \mathsf{Send}, \llbracket A \rrbracket)$$

$$\llbracket ?A \rrbracket \triangleq \lambda c.\, \mathsf{IsChan}(c, \mathsf{Recv}, \llbracket A \rrbracket)$$

Logical typing for channels

Semantic interpretation of types ("logical relation"):

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

$$\llbracket !A \rrbracket \triangleq \lambda c.\, \mathsf{IsChan}(c, \mathsf{Send}, \llbracket A \rrbracket)$$

$$\llbracket ?A \rrbracket \triangleq \lambda c.\, \mathsf{IsChan}(c, \mathsf{Recv}, \llbracket A \rrbracket)$$

The semantic typing rules for channels follow immediately from the Hoare rules

Verification of one-shot channel separation logic API in Iris

One-shot channel ownership defined using standard Iris methodology

$$\mathsf{IsChan}(c, tag, \Phi) \triangleq \ldots$$

Verification of one-shot channel separation logic API in Iris

One-shot channel ownership defined using standard Iris methodology:

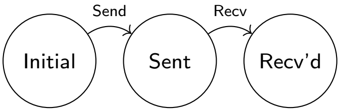
1. Model abstraction as a state transition system (STS)

$\mathsf{IsChan}(c, \mathit{tag}, \Phi) \triangleq \ldots$

Verification of one-shot channel separation logic API in Iris

One-shot channel ownership defined using standard Iris methodology:

1. Model abstraction as a state transition system (STS)

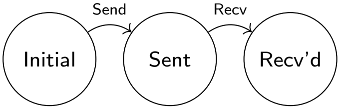


$\mathsf{IsChan}(c, \mathit{tag}, \Phi) \triangleq \ldots$

Verification of one-shot channel separation logic API in Iris

One-shot channel ownership defined using standard Iris methodology:

1. Model abstraction as a state transition system (STS)
2. Define an invariant as a disjunction of the states



$\mathsf{IsChan}(c, tag, \Phi) \triangleq \ldots$

Verification of one-shot channel separation logic API in Iris

One-shot channel ownership defined using standard Iris methodology:

1. Model abstraction as a state transition system (STS)
2. Define an invariant as a disjunction of the states

$$\text{Initial} \xrightarrow{\text{Send}} \text{Sent} \xrightarrow{\text{Recv}} \text{Recv'd}$$

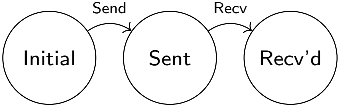
$$\mathsf{chan\_inv} \qquad c\ \Phi \triangleq (\underbrace{\qquad\qquad}_{\text{(1) initial state}}) \lor (\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad}_{\text{(2) message sent, but not yet received}}) \lor (\underbrace{\qquad\qquad\qquad}_{\text{(3) final state}})$$

$$\mathsf{IsChan}(c, \mathit{tag}, \Phi) \triangleq \ldots \qquad \boxed{\mathsf{chan\_inv} \qquad c\ \Phi}$$

Verification of one-shot channel separation logic API in Iris

One-shot channel ownership defined using standard Iris methodology:

1. Model abstraction as a state transition system (STS)
2. Define an invariant as a disjunction of the states
3. Determine resource ownership of each state



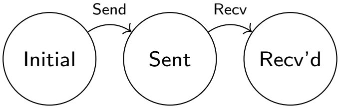
$$\mathsf{chan\_inv} \qquad c\ \Phi \triangleq (\underbrace{\qquad\qquad}_{\text{(1) initial state}}) \lor (\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad}_{\text{(2) message sent, but not yet received}}) \lor (\underbrace{\qquad\qquad\qquad}_{\text{(3) final state}})$$

$$\mathsf{IsChan}(c, tag, \Phi) \triangleq \ldots \qquad \boxed{\mathsf{chan\_inv} \qquad c\ \Phi}$$

Verification of one-shot channel separation logic API in Iris

One-shot channel ownership defined using standard Iris methodology:

1. Model abstraction as a state transition system (STS)
2. Define an invariant as a disjunction of the states
3. Determine resource ownership of each state



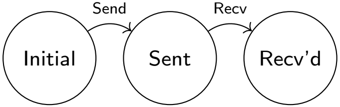
$$\mathsf{chan\_inv} \quad c\ \Phi \triangleq (\underbrace{c \mapsto \texttt{None}}_{\text{(1) initial state}}) \vee (\underbrace{\exists v.\, c \mapsto \texttt{Some}\, v \qquad\qquad}_{\text{(2) message sent, but not yet received}}) \vee (\underbrace{\qquad\qquad}_{\text{(3) final state}})$$

$$\mathsf{IsChan}(c, \mathit{tag}, \Phi) \triangleq \ldots \qquad \boxed{\mathsf{chan\_inv} \quad c\ \Phi}$$

Verification of one-shot channel separation logic API in Iris

One-shot channel ownership defined using standard Iris methodology:

1. Model abstraction as a state transition system (STS)
2. Define an invariant as a disjunction of the states
3. Determine resource ownership of each state



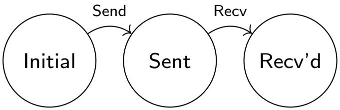
$$\mathrm{chan\_inv}\quad c\ \Phi \triangleq (\underbrace{c \mapsto \texttt{None}}_{\text{(1) initial state}}) \lor (\underbrace{\exists v.\, c \mapsto \texttt{Some}\, v * \Phi\ v \qquad}_{\text{(2) message sent, but not yet received}}) \lor (\underbrace{\qquad\qquad\qquad}_{\text{(3) final state}})$$

$$\mathrm{IsChan}(c, tag, \Phi) \triangleq \ldots \qquad \boxed{\mathrm{chan\_inv}\qquad c\ \Phi}$$

Verification of one-shot channel separation logic API in Iris

One-shot channel ownership defined using standard Iris methodology:

1. Model abstraction as a state transition system (STS)
2. Define an invariant as a disjunction of the states
3. Determine resource ownership of each state
4. Encode STS transition permissions with ghost state

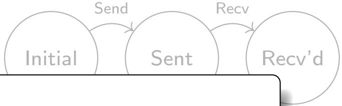


$$\text{chan\_inv} \quad c\ \Phi \triangleq (\underbrace{c \mapsto \texttt{None}}_{\text{(1) initial state}}) \vee (\underbrace{\exists v.\, c \mapsto \texttt{Some}\, v * \Phi\ v}_{\text{(2) message sent, but not yet received}}) \vee (\underbrace{\qquad\qquad}_{\text{(3) final state}})$$

$$\text{IsChan}(c, tag, \Phi) \triangleq \ldots \qquad \boxed{\text{chan\_inv} \quad c\ \Phi}$$

Verification of one-shot channel separation logic API in Iris

One-shot channel ownership defined using standard Iris methodology:



1. Model abstraction as a state transition system (STS)
2. Define an invariant as a disjunction of the states
3. Det
4. Enc

The "token" ghost state:

$$\begin{array}{rcl} \mathsf{True} & \Rrightarrow\!\!\ast & \exists \gamma.\, \mathsf{tok}\,\gamma \\ \mathsf{tok}\,\gamma * \mathsf{tok}\,\gamma & \twoheadrightarrow\!\!\ast & \mathsf{False} \end{array}$$

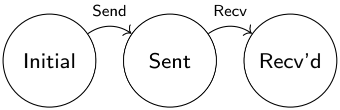
$$\mathsf{chan\_inv}\ \ c\ \Phi \triangleq \underbrace{\left(\ c \mapsto \mathtt{None}\ \right)}_{\text{(1) initial state}} \vee \underbrace{\left(\exists v.\, c \mapsto \mathtt{Some}\ v * \Phi\ v\ \right)}_{\text{(2) message sent, but not yet received}} \vee \underbrace{\left(\qquad\qquad\right)}_{\text{(3) final state}}$$

$$\mathsf{IsChan}(c, \mathit{tag}, \Phi) \triangleq \ldots \qquad \boxed{\mathsf{chan\_inv}\qquad c\ \Phi}$$

Verification of one-shot channel separation logic API in Iris

One-shot channel ownership defined using standard Iris methodology:

1. Model abstraction as a state transition system (STS)
2. Define an invariant as a disjunction of the states
3. Determine resource ownership of each state
4. Encode STS transition permissions with ghost state



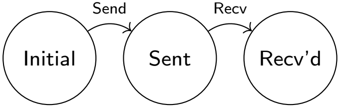
$$\mathrm{chan\_inv} \quad c\ \Phi \triangleq (\underbrace{c \mapsto \texttt{None}}_{\text{(1) initial state}}) \vee (\underbrace{\exists v.\, c \mapsto \texttt{Some}\, v * \Phi\ v}_{\text{(2) message sent, but not yet received}}) \vee (\underbrace{\qquad\qquad}_{\text{(3) final state}})$$

$$\mathrm{IsChan}(c, tag, \Phi) \triangleq \ldots \qquad \boxed{\mathrm{chan\_inv} \quad c\ \Phi}$$

Verification of one-shot channel separation logic API in Iris

One-shot channel ownership defined using standard Iris methodology:

1. Model abstraction as a state transition system (STS)
2. Define an invariant as a disjunction of the states
3. Determine resource ownership of each state
4. Encode STS transition permissions with ghost state



$$\mathsf{chan\_inv}\ \gamma_s\ \gamma_r\ c\ \Phi \triangleq (\underbrace{c \mapsto \texttt{None}}_{\text{(1) initial state}}) \lor (\underbrace{\exists v.\, c \mapsto \texttt{Some}\, v * \Phi\ v * \mathsf{tok}\,\gamma_s}_{\text{(2) message sent, but not yet received}}) \lor (\underbrace{\mathsf{tok}\,\gamma_s * \mathsf{tok}\,\gamma_r}_{\text{(3) final state}})$$

$$\mathsf{IsChan}(c, \mathit{tag}, \Phi) \triangleq \exists \gamma_s, \gamma_r.\, \boxed{\mathsf{chan\_inv}\ \gamma_s\ \gamma_r\ c\ \Phi} * \begin{cases} \mathsf{tok}\,\gamma_s & \text{if } \mathit{tag} = \mathsf{Send} \\ \mathsf{tok}\,\gamma_r & \text{if } \mathit{tag} = \mathsf{Recv} \end{cases}$$

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

Perennial DimSum Cerise RustBelt ReLoC

Melocoton VMSL RefinedC Aneris

Diaframe Iris-Wasm

Iris

Simuliris Compass

Iris-Tini Affect RustHornBelt

Cosmo CQS SeLoC

iGPS Hazel gDOT GoJournal

OCPL Islaris Actris Iron iRC11

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

Future work: Going beyond safety

- App… liver…
- Diff… (ins… …r
- We…
- But… …ularly
- Nor…

I aim to address these challenges in my ERC Consolidator project (2025-2030) and in our NWO XL project

Developing Correct Concurrent Software Using Types (COCONUT)

Looking for a PhD student (start date: before October 2026) and 2 postdocs (start date: 2027)

`https://robbertkrebbers.nl/coconut.html`

Read more?

Our overview:

A Logical Approach to Type Soundness

AMIN TIMANY, Aarhus University, Denmark
ROBBERT KREBBERS, Radboud University Nijmegen, The Netherlands
DEREK DREYER, MPI-SWS, Germany
LARS BIRKEDAL, Aarhus University, Denmark

Type soundness, which asserts that "well-typed programs cannot go wrong", is widely viewed as the canonical theorem one must prove to establish that a type system is doing its job. It is commonly proved using the so-called *syntactic approach* (aka *progress and preservation*), which has had a huge impact on the study and teaching of programming language foundations. Unfortunately, syntactic type soundness is a rather weak theorem. It only applies to programs that are completely well-typed, and thus tells us nothing about the many programs written in "safe" languages that make use of "unsafe" language features. Even worse, it tells us nothing about whether type systems achieve one of their main goals: enforcement of data abstraction. One can easily define a language that enjoys syntactic type soundness and yet fails to support even the most basic modular reasoning principles for abstraction mechanisms like closures, objects, and abstract data types.

In this paper, we argue that we should no longer be satisfied with just proving syntactic type soundness, and should instead start proving a stronger theorem—*semantic type soundness*—which captures more accurately what type systems are actually good for. Semantic type soundness is an old idea—Milner's original formulation of type soundness was a semantic one—but it fell out of favor in the 1990s due to limitations and complexities of denotational models. In the succeeding decades, thanks to a series of technical advances—notably, (1) *step-indexed Kripke logical relations* constructed over operational semantics and (2) *higher-order concurrent separation logic* as consolidated in the *Iris* framework in Coq—we can now build (machine-checked) semantic soundness proofs at a much higher level of abstraction than was previously possible.

The resulting "logical" approach to semantic type soundness has already been employed to great effect in a number of recent papers (by us and others), but those papers typically concern advanced problem scenarios that complicate the presentation, they assume significant prior knowledge of the reader, and they refrain from giving many details of the proofs. Here, we hope to provide a gentler, more pedagogically motivated introduction to logical type soundness, aimed at a broader audience that may or may not be familiar with logical relations and Iris. As a bonus, we also show how logical type soundness proofs can be easily generalized to establish an even stronger *relational* property—*representation independence*—for realistic type systems.

> Type structure is a syntactic discipline for enforcing levels of abstraction.
> – Reynolds [1983]

> Although types and assertions may be semantically similar, the actual development of type systems for programming languages has been quite separate from the development of approaches to specification such as Hoare logic…the real question is whether the dividing line between types and assertions can be erased. – Reynolds [2002]

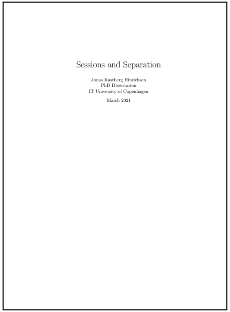
This paper is dedicated to the memory of John C. Reynolds.

1 INTRODUCTION

The *type soundness* (or *type safety*) theorem for a programming language states that if a program in that language passes the type checker, it should be guaranteed to have well-defined behavior when executed. Introduced over 40 years ago by Milner [1978], type soundness has become the canonical property that type systems for "safe" programming languages are expected to satisfy.

Authors' addresses: Amin Timany, Aarhus University, Denmark, timany@cs.au.dk; Robbert Krebbers, Radboud University Nijmegen, The Netherlands, mail@robbertkrebbers.nl; Derek Dreyer, MPI-SWS, Saarland Informatics Campus, Germany, dreyer@mpi-sws.org; Lars Birkedal, Aarhus University, Denmark, birkedal@cs.au.dk.

Session types:

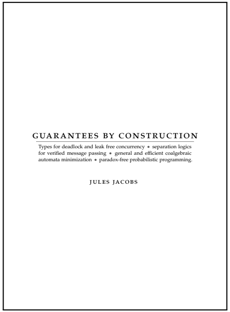


Sessions and Separation

Jonas Kastberg Hinrichsen
PhD Dissertation
IT University of Copenhagen

March 2021

Deadlock freedom:

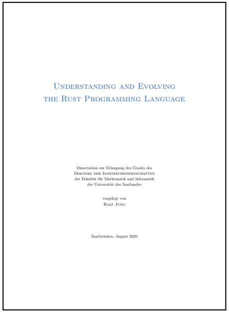


GUARANTEES BY CONSTRUCTION

Types for deadlock and leak free concurrency • separation logics for verified message passing • general and efficient coalgebraic automata minimization • paradox-free probabilistic programming.

JULES JACOBS

Rust:



UNDERSTANDING AND EVOLVING THE RUST PROGRAMMING LANGUAGE

Dissertation zur Erlangung des Grades des
Doktors der Ingenieurwissenschaften
der Fakultät für Mathematik und Informatik
der Universität des Saarlandes

vorgelegt von
Ralf Jung

Saarbrücken, August 2020

https://iris-project.org

The logical approach in Iris crucially depends on using separation logic as a meta theory: both to prove the fundamental theorem and to verify “unsafe” code

How to do mechanized proofs in separation logic?

What is Iris?

1. Iris Proof Mode (IPM)
   Tactic language for separation logic in Rocq
2. Iris Theory
   Building blocks for developing your own concurrent separation logic
3. Iris HeapLang
   The default language shipped with Iris's Rocq development

How is Iris used?

Developing a logic: Use Iris as a meta theory to develop a separation logic

Deploying a logic: Verify programs or a type system using the developed logic

How is Iris used?

Developing a logic: Use Iris as a meta theory to develop a separation logic

- for a specific language: HeapLang, Rust, C, Go, WASM, capability machines, ...
- program property: functional correctness, non-interference, crash safety, refinement, complexity, ...
- programming paradigm: algebraic effects, distributed systems, session types, relaxed memory concurrency, ...
- depending on the desired logic, one can use different building blocks of Iris

Deploying a logic: Verify programs or a type system using the developed logic

How is Iris used?

Developing a logic: Use Iris as a meta theory to develop a separation logic

- for a specific language: HeapLang, Rust, C, Go, WASM, capability machines, ...
- program property: functional correctness, non-interference, crash safety, refinement, complexity, ...
- programming paradigm: algebraic effects, distributed systems, session types, relaxed memory concurrency, ...
- depending on the desired logic, one can use different building blocks of Iris

Deploying a logic: Verify programs or a type system using the developed logic

For both developing and deploying logics,
a proof assistant is essential

Wanted:

proof assistant for

Iris

Wanted:

proof assistant for

Iris

Very different from the logic of Rocq/HOL/etc

Wanted:

proof assistant for
higher-order
impredicative
modal
concurrent
separation logic

Very different from the logic of Rocq/HOL/etc

How?

Embed proof assistant in existing proof assistant

How?

Embed proof assistant in existing proof assistant

Why?

Prove soundness of embedded proof assistant
Reuse infrastructure of host proof assistant
Users do not need to learn new tool

How to do proofs in separation logic

Suppose we want to prove $P * (\exists a.\, \Phi a) * Q \;\vdash\; Q * (\exists a.\, P * \Phi a)$

How to do proofs in separation logic

Suppose we want to prove $P * (\exists a.\, \Phi a) * Q \;\vdash\; Q * (\exists a.\, P * \Phi a)$

1. **Unfold definitions of the model**: $\forall \sigma.\, (\exists \sigma_1\, \sigma_2.\, \sigma = \sigma_1 \uplus \sigma_2 \wedge P \sigma_1 \wedge \ldots) \to \ldots$
   - Defeats the purpose of separation logic to hide reasoning about disjointness
   - Does not scale to larger goals or modal models

How to do proofs in separation logic

Suppose we want to prove $P * (\exists a.\, \Phi a) * Q \;\vdash\; Q * (\exists a.\, P * \Phi a)$

1. **Unfold definitions of the model**: $\forall \sigma.\, (\exists \sigma_1\, \sigma_2.\, \sigma = \sigma_1 \uplus \sigma_2 \wedge P \sigma_1 \wedge \ldots) \rightarrow \ldots$
   - Defeats the purpose of separation logic to hide reasoning about disjointness
   - Does not scale to larger goals or modal models
2. **Use the laws of separation logic**: associativity/commutativity of $*$, distributivity of $\exists$ over $*$, ...
   - Too low-level, already small proofs require many steps
   - Also rather slow

How to do proofs in separation logic

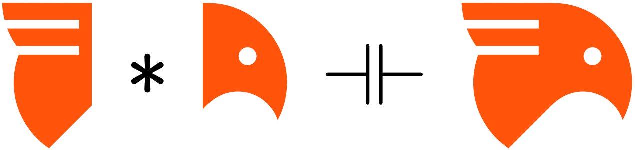
Suppose we want to prove $P * (\exists a.\, \Phi a) * Q \;\vdash\; Q * (\exists a.\, P * \Phi a)$

1. **Unfold definitions of the model**: $\forall \sigma.\, (\exists \sigma_1\, \sigma_2.\, \sigma = \sigma_1 \uplus \sigma_2 \wedge P \sigma_1 \wedge \ldots) \rightarrow \ldots$
   - Defeats the purpose of separation logic to hide reasoning about disjointness
   - Does not scale to larger goals or modal models
2. **Use the laws of separation logic**: associativity/commutativity of $*$, distributivity of $\exists$ over $*$, ...
   - Too low-level, already small proofs require many steps
   - Also rather slow
3. **Use Iris**

Iris Proof Mode (IPM) [Krebbers *et al.*; POPL'17, ICFP'18]

Enable tactic-style proofs in separation logic

- Extend Rocq with named proof contexts for separation logic
- Tactics for introduction and elimination of all connectives of separation logic ...
- ...that can be used in Rocq's mechanisms for automation/tactic programming
- Implemented without modifying Rocq (using reflection, type classes and Ltac)



Iris Proof Mode demo

```
Lemma test {A} (P Q : iProp) (Φ : A → iProp) :
  P ∗ (∃ a, Φ a) ∗ Q ⊢ Q ∗ ∃ a, P ∗ Φ a.
Proof.
  iIntros "[H1 [H2 H3]]".
  iDestruct "H2" as (x) "H2".
  iSplitL "H3".
  - iAssumption.
  - iExists x.
    iFrame.
Qed.
```

Iris Proof Mode demo

```
Lemma test {A} (P Q : iProp) (Φ : A → iProp) :
  P ∗ (∃ a, Φ a) ∗ Q ⊢ Q ∗ ∃ a, P ∗ Φ a.
Proof.
  i
  iDestruct "H2" as (x) "H2".
  iSplitL "H3".
  - iAssumption.
  - iExists x.
    iFrame.
Qed.
```

Lemma in separation logic

Iris Proof Mode demo

```
Lemma test {A} (P Q : iProp) (Φ : A → iProp) :
  P ∗ (∃ a, Φ a) ∗ Q ⊢ Q ∗ ∃ a, P ∗ Φ a.
Proof.
  iIntros "[H1 [H2 H3]]".
  iDestruct "H2" as (x) "H2".
  iSplitL "H3".
  - iAssumption.
  - iExists x.
    iFrame.
Qed.
```

```
1 subgoal
A : Type
P, Q : iProp
Φ : A → iProp
______________________________________(1/1)
P ∗ (∃ a : A, Φ a) ∗ Q
⊢ Q ∗ (∃ a : A, P ∗ Φ a)
```

Iris Proof Mode demo

```
Lemma test {A} (P Q : iProp) (Φ : A → iProp) :
  P ∗ (∃ a, Φ a) ∗ Q ⊢ Q ∗ ∃ a, P ∗ Φ a.
Proof.
  iIntros "[H1 [H2 H3]]".
  iDestruct "H2" as (x) "H2".
  iSplitL "H3".
  - iAssumption.
  - iExists x.
    iFrame.
Qed.
```

```
1 subgoal
A : Type
P, Q : iProp
Φ : A → iProp
______________________(1/1)
"H1" : P
"H2" : ∃ a : A, Φ a
"H3" : Q
______________________∗
Q ∗ (∃ a : A, P ∗ Φ a)
```

Iris Proof Mode demo

```
Lemma test {A} (P Q : iProp) (Φ : A → iProp) :
  P ∗ (∃ a, Φ a) ∗ Q ⊢ Q ∗ ∃ a, P ∗ Φ a.
Proof.
  iIntros "[H1 [H2 H3]]".
  iDestruct "H2" as (x) "H2".
  iSplitL "H3".
  - iAssumption.
  - iExists x.
    iFrame.
Qed.
```

```
1 subgoal
A : Type
P, Q : iProp
Φ : A → iProp
x : A
______________________________________(1/1)
"H1" : P
"H2" : Φ x
"H3" : Q
--------------------------------------∗
Q ∗ (∃ a : A, P ∗ Φ a)
```

Iris Proof Mode demo

```
Lemma test {A} (P Q : iProp) (Φ : A → iProp) :
  P ∗ (∃ a, Φ a) ∗ Q ⊢ Q ∗ ∃ a, P ∗ Φ a.
Proof.
  iIntros "[H1 [H2 H3]]".
  iDestruct "H2" as (x) "H2".
  iSplitL "H3".
  - iAssumption.
  - iExists x.
    iFrame.
Qed.
```

```
1 subgoal
A : Type
P, Q : iProp
Φ : A → iProp
x : A
______________________________________(1/1)
"H1" : P
"H2" : Φ x
"H3" : Q
--------------------------------------∗
Q ∗ (∃ a : A, P ∗ Φ a)
```

∗ means: resources should be split

Iris Proof Mode demo

```
Lemma test {A} (P Q : iProp) (Φ : A → iProp) :
  P ∗ (∃ a, Φ a) ∗ Q ⊢ Q ∗ ∃ a, P ∗ Φ a.
Proof.
  iIntros "[H1 [H2 H3]]".
  iDestruct "H2" as (x) "H2".
  iSplitL "H3".
  - iAssumption.
```

The hypotheses for the left conjunct

```
Qed.
```

```
1 subgoal
A : Type
P, Q : iProp
Φ : A → iProp
x : A
______________________(1/1)
"H1" : P
"H2" : Φ x
"H3" : Q
----------------------∗
Q ∗ (∃ a : A, P ∗ Φ a)
```

∗ means: resources should be split

Iris Proof Mode demo

```
Lemma test {A} (P Q : iProp) (Φ : A → iProp) :
  P ∗ (∃ a, Φ a) ∗ Q ⊢ Q ∗ ∃ a, P ∗ Φ a.
Proof.
  iIntros "[H1 [H2 H3]]".
  iDestruct "H2" as (x) "H2".
  iSplitL "H3".
  - iAssumption.
  - iExists x.
    iFrame.
Qed.
```

```
2 subgoals
A : Type
P, Q : iProp
Φ : A → iProp
x : A
______________________________________(1/2)
"H3" : Q
--------------------------------------∗
Q

______________________________________(2/2)
"H1" : P
"H2" : Φ x
--------------------------------------∗
∃ a : A, P ∗ Φ a
```

Iris Proof Mode demo

```
Lemma test {A} (P Q : iProp) (Φ : A → iProp) :
  P ∗ (∃ a, Φ a) ∗ Q ⊢ Q ∗ ∃ a, P ∗ Φ a.
Proof.
  iIntros "[H1 [H2 H3]]".
  by iFrame.
Qed.
```

We can also solve this goal automatically

```
1 subgoal
A : Type
P, Q : iProp
Φ : A → iProp
x : A
______________________________________(1/1)
"H1" : P
"H2" : ∃ a, Φ a
"H3" : Q
--------------------------------------∗
Q ∗ (∃ a : A, P ∗ Φ a)
```

Iris Proof Mode demo

```
Lemma test {A} (P Q : iProp) (Φ : A → iProp) :
  P ∗ (∃ a, Φ a) ∗ Q ⊢ Q ∗ ∃ a, P ∗ Φ a.
Proof.
  iIntros "[H1 [H2 H3]]".
  by iFrame.
Qed.
```

We can also solve this goal automatically

```
No more subgoals.
```

Iris Proof Mode demo

```
Lemma test {A} (P Q : iProp) (Φ : A → iProp) :
  P ∗ (∃ a, Φ a) ∗ Q ⊢ Q ∗ ∃ a, P ∗ Φ a.
Proof.
  iIntros "[$ [? $]] //".
Qed.
```

Or use intro patterns

Features of the Iris Proof Mode

- **Proofs have the look and feel of ordinary Rocq proofs**
  For many Rocq tactics `tac`, we have a variant `iTac`

Features of the Iris Proof Mode

- **Proofs have the look and feel of ordinary Rocq proofs**
  For many Rocq tactics `tac`, we have a variant `iTac`
- **Support for advanced features of separation logic**
  Higher-order quantification, modalities, invariants, ghost state, ...

Features of the Iris Proof Mode

- **Proofs have the look and feel of ordinary Rocq proofs**
  For many Rocq tactics `tac`, we have a variant `iTac`
- **Support for advanced features of separation logic**
  Higher-order quantification, modalities, invariants, ghost state, ...
- **Integration with tactics for proving programs**
  Symbolic execution tactics for weakest preconditions

Features of the Iris Proof Mode

- **Proofs have the look and feel of ordinary Rocq proofs**
  For many Rocq tactics `tac`, we have a variant `iTac`
- **Support for advanced features of separation logic**
  Higher-order quantification, modalities, invariants, ghost state, ...
- **Integration with tactics for proving programs**
  Symbolic execution tactics for weakest preconditions
- **Tactic programming**
  One can combine/program with IPM tactics using Rocq's Ltac like ordinary Rocq tactics

Implementation of Iris Proof Mode

How to embed a logic into a proof assistant?

Deep embedding	Shallow embedding

`Inductive form : Type :=`
`  ∣ iAnd : form → form → form`
`  ∣ iForall : string → form → form → form`
	`Definition iProp : Type :=`
	`  (* fancy "predicates over states" *).`
	`Definition iAnd : iProp → iProp → iProp :=`
	`  (* semantic interpretation *).`
	`Definition iForall : ∀ A, (A → iProp) → iProp :=`
	`  (* semantic interpretation *).`

How to embed a logic into a proof assistant?

Deep embedding	**Shallow embedding**

`Inductive form : Type :=`
`  | iAnd : form → form → form`
`  | iForall : string → form → form → form`
	`Definition iProp : Type :=`
	`  (* fancy "predicates over states" *).`
	`Definition iAnd : iProp → iProp → iProp :=`
	`  (* semantic interpretation *).`
	`Definition iForall : ∀ A, (A → iProp) → iProp :=`
	`  (* semantic interpretation *).`

Traverse formulas using Coq functions (fast)	Traverse formulas on the meta level (slow)
Reflective tactics (fast)	Tactics on the meta level (slow)

How to embed a logic into a proof assistant?

Deep embedding	Shallow embedding
`Inductive form : Type := | iAnd : form → form → form | iForall : string → form → form → form`	`Definition iProp : Type := (* fancy "predicates over states" *). Definition iAnd : iProp → iProp → iProp := (* semantic interpretation *). Definition iForall : ∀ A, (A → iProp) → iProp := (* semantic interpretation *).`
Traverse formulas using Coq functions (fast)	Traverse formulas on the meta level (slow)
Reflective tactics (fast)	Tactics on the meta level (slow)
Need to explicitly encode binders	Reuse binders of Coq
Need to embed features such as lists	Piggy-back on features such as lists from Coq

Deep embedding:

```
Inductive form : Type :=
  | iAnd : form → form → form
  | iForall : string → form → form → form
```

Shallow embedding:

```
Definition iProp : Type :=
  (* fancy "predicates over states" *).
Definition iAnd : iProp → iProp → iProp :=
  (* semantic interpretation *).
Definition iForall : ∀ A, (A → iProp) → iProp :=
  (* semantic interpretation *).
```

How to embed a logic into a proof assistant?

Deep embedding	**Shallow embedding**

`Inductive form : Type :=`
`  | iAnd : form → form → form`
`  | iForall : string → form → form → form`
	`Definition iProp : Type :=`
	`  (* fancy "predicates over states" *).`
	`Definition iAnd : iProp → iProp → iProp :=`
	`  (* semantic interpretation *).`
	`Definition iForall : ∀ A, (A → iProp) → iProp :=`
	`  (* semantic interpretation *).`

Traverse formulas using Coq functions (fast)	Traverse formulas on the meta level (slow)
Reflective tactics (fast)	Tactics on the meta level (slow)
Need to explicitly encode binders	Reuse binders of Coq
Need to embed features such as lists	Piggy-back on features such as lists from Coq
Grammar of formulas fixed once and forall	Easily extensible with new connectives

How to embed a logic into a proof assistant?

Deep embedding	**Shallow embedding**

`Inductive form : Type :=`
`  | iAnd : form → form → form`
`  | iForall : string → form → form → form`
	`Definition iProp : Type :=`
	`  (* fancy "predicates over states" *).`
	`Definition iAnd : iProp → iProp → iProp :=`
	`  (* semantic interpretation *).`
	`Definition iForall : ∀ A, (A → iProp) → iProp :=`
	`  (* semantic interpretation *).`

Traverse formulas using Coq functions (fast)	Traverse formulas on the meta level (slow)
Reflective tactics (fast)	Tactics on the meta level (slow)
Need to explicitly encode binders	Reuse binders of Coq
Need to embed features such as lists	Piggy-back on features such as lists from Coq
Grammar of formulas fixed once and forall	Easily extensible with new connectives

Context manipulation is the prime task of tactics:
Deeply embedded contexts, shallowly embedded logic ⇒ Best of both worlds

Deeply embedded contexts (1)

```
Lemma test {A} (P Q : iProp) (Φ : A → iProp) :
  P ∗ (∃ a, Φ a) ∗ Q ⊢ Q ∗ ∃ a, P ∗ Φ a.
Proof.
  iIntros "[H1 [H2 H3]]".
  iDestruct "H2" as (x) "H2".
  iSplitL "H3".
  - iAssumption.
  - iExists x.
    iFrame.
Qed.
```

Deeply embedded contexts (1)

```
Lemma test {A} (P Q : iProp) (Φ : A → iProp) :
  P ∗ (∃ a, Φ a) ∗ Q ⊢ Q ∗ ∃ a, P ∗ Φ a.
Proof.
  iIntros "[H1 [H2 H3]]".
  iDestruct "H2" as (x) "H2".
  iSplitL "H3".
  - iAssumption.
  - iExists x.
    iFrame.
Qed.
```

```
1 subgoal
A : Type
P, Q : iProp
Φ : A → iProp
x : A
______________________________________(1/1)
"H1" : P
"H2" : Φ x
"H3" : Q
--------------------------------------∗
Q ∗ (∃ a : A, P ∗ Φ a)
```

Deeply embedded contexts (1)

```
Lemma test {A} (P Q : iProp) (Φ : A → iProp) :
  P ∗ (∃ a, Φ a) ∗ Q ⊢ Q ∗ ∃ a, P ∗ Φ a.
Proof.
  iIntros "[H1 [H2 H3]]".
  iDestruct "H2" as (x) "H2".
  Unset Printing Notations.
```

```
1 subgoal
A : Type
P, Q : iProp
Φ : A → iProp
x : A
______________________(1/1)
"H1" : P
"H2" : Φ x
                     ______∗
Q ∗ (∃ a : A, P ∗ Φ a)
```

Notation for deeply embedded context

Deeply embedded contexts (1)

```
Lemma test {A} (P Q : iProp) (Φ : A → iProp) :
  P ∗ (∃ a, Φ a) ∗ Q ⊢ Q ∗ ∃ a, P ∗ Φ a.
Proof.
  iIntros "[H1 [H2 H3]]".
  iDestruct "H2" as (x) "H2".
  Unset Printing Notations.
```

```
1 subgoal
A : Type
P, Q : iProp
Φ : A → iProp
x : A
______________________(1/1)
envs_entails (Envs Enil
 (Esnoc (Esnoc (Esnoc Enil
   (String (Ascii false
     false false true false
     false true false)
   (String (Ascii true
     false false false true
     true false false)
   EmptyString)) P)
   ...
```

Deeply embedded contexts (2)

Visible goal (with pretty printing):

$$
\begin{array}{l}
\vec{\mathtt{x}} \;:\; \vec{\phi} \qquad \text{Variables and pure Coq hypotheses} \\
\text{---------------------------------------} \\
\Pi \qquad \text{Spatial separation logic hypotheses} \\
\hline\text{}\hspace{18em}{*} \\
Q \qquad \text{Separation logic goal}
\end{array}
$$

Deeply embedded contexts (2)

Visible goal (with pretty printing):

$\vec{x} : \vec{\phi}$ Variables and pure Coq hypotheses

Π Spatial separation logic hypotheses

Q Separation logic goal

Actual Coq goal (without pretty printing):

$\vec{x} : \vec{\phi}$

$\Pi \Vdash Q$

Where:

$$P_1, \ldots, P_n \Vdash Q \triangleq (P_1 * \cdots * P_n) \vdash Q$$

Implementation of the `iSplitL`/`iSplitR` tactic (simplified)

Tactics implemented by reflection as mere lemmas:

```
Lemma tac_sep_split Π Π₁ Π₂ Hs Q₁ Q₂ :
  envs_split Hs Π = Some (Π₁,Π₂) →
  (Π₁ ⊩ Q₁) → (Π₂ ⊩ Q₂) → Π ⊩ Q₁ ∗ Q₂.
```

$$\frac{\Pi_1 \Vdash Q_1 \qquad \Pi_2 \Vdash Q_2}{\Pi_1, \Pi_2 \Vdash Q_1 * Q_2}$$

Implementation of the `iSplitL`/`iSplitR` tactic (simplified)

Tactics implemented by reflection as mere lemmas:

```
Lemma tac_sep_split Π Π₁ Π₂ Hs Q₁ Q₂ :
  envs_split Hs Π = Some (Π₁,Π₂) →
  (Π₁ ⊩ Q₁) → (Π₂ ⊩ Q₂) → Π ⊩ Q₁ ∗ Q₂.
```

$$\frac{\Pi_1 \Vdash Q_1 \qquad \Pi_2 \Vdash Q_2}{\Pi_1, \Pi_2 \Vdash Q_1 * Q_2}$$

Context splitting implemented as a computable Coq function

Implementation of the `iSplitL`/`iSplitR` tactic (simplified)

Tactics implemented by reflection as mere lemmas:

```
Lemma tac_sep_split Π Π₁ Π₂ Hs Q₁ Q₂ :
  envs_split Hs Π = Some (Π₁,Π₂) →
  (Π₁ ⊩ Q₁) → (Π₂ ⊩ Q₂) → Π ⊩ Q₁ ∗ Q₂.
```

$$\frac{\Pi_1 \Vdash Q_1 \qquad \Pi_2 \Vdash Q_2}{\Pi_1, \Pi_2 \Vdash Q_1 * Q_2}$$

Context splitting implemented as a computable Coq function

Ltac wrappers around the reflective tactic:

```
Tactic Notation "iSplitL" constr(Hs) :=
  let Hs := words Hs in
  eapply tac_sep_split with _ _ Hs _ _;
    [ pm_reflexivity || fail "iSplitL: hypotheses" Hs "not found"
    | (* goal 1 *)
    | (* goal 2 *) ].
```

Implementation of the iSplitL/iSplitR tactic (simplified)

Tactics implemented by reflection as mere lemmas:

```
Lemma tac_sep_split Π Π₁ Π₂ Hs Q₁ Q₂ :
  envs_split Hs Π = Some (Π₁,Π₂) →
  (Π₁ ⊩ Q₁) → (Π₂ ⊩ Q₂) → Π ⊩ Q₁ ∗ Q₂.
```

$$\frac{\Pi_1 \Vdash Q_1 \qquad \Pi_2 \Vdash Q_2}{\Pi_1, \Pi_2 \Vdash Q_1 * Q_2}$$

Context splitting implemented as a computable Coq function

Ltac wrappers around the reflective tactic:

```
Tactic Notation "iSplitL" constr(Hs) :=
  let Hs := words Hs in
  eapply tac_sep_split with _ _ Hs _ _;
    [ pm_reflexivity || fail "iSplitL: hypotheses" Hs "not found"
    | (* goal 1 *)
    | (* goal 2 *) ].
```

Proof is just eq_refl

Implementation of the `iSplitL`/`iSplitR` tactic (simplified)

Tactics implemented by reflection as mere lemmas:

```
Lemma tac_sep_split Π Π₁ Π₂ Hs Q₁ Q₂ :
  envs_split Hs Π = Some (Π₁,Π₂) →
  (Π₁ ⊩ Q₁) → (Π₂ ⊩ Q₂) → Π ⊩ Q₁ ∗ Q₂.
```

$$\frac{\Pi_1 \Vdash Q_1 \qquad \Pi_2 \Vdash Q_2}{\Pi_1, \Pi_2 \Vdash Q_1 * Q_2}$$

Context splitting implemented as a computable Coq function

Ltac wrappers around the reflective tactic:

```
Tactic Notation "iSplitL" constr(Hs) :=
  let Hs := words Hs in
  eapply tac_sep_split with _ _ Hs _ _;
    [ pm_reflexivity || fail "iSplitL: hypotheses" Hs "not found"
    | (* goal 1 *)
    | (* goal 2 *) ].
```

Report sensible error to the user

Proof is just `eq_refl`

Implementation of the `iFrame` tactic (1) (simplified)

$$\frac{\Pi \Vdash Q \qquad Q \text{ is } P \text{ with } R \text{ canceled}}{\Pi, R \Vdash P}$$

Implementation of the `iFrame` tactic (1) (simplified)

$$\frac{\Pi \Vdash Q \qquad Q \text{ is } P \text{ with } R \text{ canceled}}{\Pi, R \Vdash P}$$

Problem: Propositions (P, Q, R) are shallow embedded, cannot `match` on them
Solution: Transform P into Q using logic programming with type classes

Implementation of the `iFrame` tactic (1) (simplified)

$$\frac{\Pi \Vdash Q \qquad Q \text{ is } P \text{ with } R \text{ canceled}}{\Pi, R \Vdash P}$$

Problem: Propositions (P, Q, R) are shallow embedded, cannot `match` on them
Solution: Transform P into Q using logic programming with type classes

```
Class Frame (R P Q : iProp) := frame : R ∗ Q ⊢ P.
```

What we want to frame

Conclusion of the new goal in which R is framed

Initial conclusion

```
Lemma tac_frame Δ Δ' i p R P Q :
  envs_lookup_delete i Δ = Some (R, Δ') →
  Frame R P Q →
  (Δ' ⊩ Q) → Δ ⊩ P.
```

Implementation of the `iFrame` tactic (1) (simplified)

$$\frac{\Pi \Vdash Q \qquad Q \text{ is } P \text{ with } R \text{ canceled}}{\Pi, R \Vdash P}$$

Problem: Propositions (P, Q, R) are shallow embedded, cannot `match` on them
Solution: Transform P into Q using logic programming with type classes

```
Class Frame (R P Q : iProp) := frame : R ∗ Q ⊢ P.
```

What we want to frame

Initial conclusion

Conclusion of the new goal in which `R` is framed

```
Lemma tac_frame Δ Δ' i p R P Q :
  envs_lookup_delete i Δ = Some (R, Δ') →
  Frame R P Q →
  (Δ' ⊩ Q) → Δ ⊩ P.
```

Implementation of the `iFrame` tactic (1) (simplified)

$$\frac{\Pi \Vdash Q \qquad Q \text{ is } P \text{ with } R \text{ canceled}}{\Pi, R \Vdash P}$$

Problem: Propositions (P, Q, R) are shallow embedded, cannot `match` on them
Solution: Transform P into Q using logic programming with type classes

```
Class Frame (R P Q : iProp) := frame : R ∗ Q ⊢ P.
```

What we want to frame

Conclusion of the new goal in which `R` is framed

Initial conclusion

```
Lemma tac_frame Δ Δ' i p R P Q :
  envs_lookup_delete i Δ = Some (R, Δ') →
  Frame R P Q →
  (Δ' ⊩ Q) → Δ ⊩ P.
```

Note: we support framing under binders (∃, ∀, . . .) and user-defined connectives

Implementation of the `iFrame` tactic (2) (simplified)

```
Class Frame (R P Q : iProp) := frame : R ∗ Q ⊢ P.
```

What we want to frame

Conclusion of the new goal in which `R` is framed

Initial conclusion

Implementation of the iFrame tactic (2) (simplified)

```
Class Frame (R P Q : iProp) := frame : R ∗ Q ⊢ P.
```

What we want to frame

Conclusion of the new goal in which R is framed

Initial conclusion

Instances (rules of the logic program):

```
Instance frame_here R : Frame R R True.
Instance frame_sep_l R P₁ P₂ Q :
  Frame R P₁ Q → Frame R (P₁ ∗ P₂) (Q ∗ P₂).
Instance frame_sep_r R P₁ P₂ Q :
  Frame R P₂ Q → Frame R (P₁ ∗ P₂) (P₁ ∗ Q).
```

Implementation of the iFrame tactic (2) (simplified)

```
Class Frame (R P Q : iProp) := frame : R ∗ Q ⊢ P.
```

What we want to frame

Conclusion of the new goal in which R is framed

Initial conclusion

Instances (rules of the logic program):

```
Class MakeSep P Q PQ := make_sep : P ∗ Q ⊣⊢ PQ.
Instance frame_here R : Frame R R emp.
Instance frame_sep_l R P₁ P₂ Q Q' :
  Frame R P₁ Q → MakeSep Q P₂ Q' → Frame R (P₁ ∗ P₂) Q'.
Instance frame_sep_r R P₁ P₂ Q Q' :
  Frame R P₂ Q → MakeSep P₁ Q Q' → Frame R (P₁ ∗ P₂) Q'.

(** Clean spurious [emp]s *)
Instance make_sep_true_l P : MakeSep emp P P | 1.
Instance make_sep_true_r P : MakeSep P emp P | 1.
Instance make_sep_default P Q : MakeSep P Q (P ∗ Q) | 2.
```

Making Iris Proof Mode parametric in the separation logic (1)

Proofs in a specific logic:

```
Lemma test {A} (P Q : iProp) (Φ : A → iProp) :
  P ∗ (∃ a, Φ a) ∗ Q ⊢ Q ∗ ∃ a, P ∗ Φ a.
Proof.
  iIntros "[H1 [H2 H3]]".
  iDestruct "H2" as (x) "H2".
  iSplitL "H3".
  – iAssumption.
  – iExists x.
    iFrame.
Qed.
```

Proofs for all logics:

```
Lemma test {PROP : bi} {A} (P Q : PROP) (Φ : A → PROP) :
  P ∗ (∃ a, Φ a) ∗ Q ⊢ Q ∗ ∃ a, P ∗ Φ a.
Proof.
  iIntros "[H1 [H2 H3]]".
  iDestruct "H2" as (x) "H2".
  iSplitL "H3".
  – iAssumption.
  – iExists x.
    iFrame.
Qed.
```

Making Iris Proof Mode parametric in the separation logic (1)

Proofs in a specific logic:

```
Lemma test {A} (P Q : iProp) (Φ : A → iProp) :
  P ∗ (∃ a, Φ a) ∗ Q ⊢ Q ∗ ∃ a, P ∗ Φ a.
Proof.
  iIntros "[H1 [H2 H3]]".
  iDestruct "H2" as (x) "H2".
  iSplitL "H3".
  – iAssumption.
  – iExists x.
    iFrame.
Qed.
```

Proofs for all logics:

```
Lemma test {PROP : bi} {A} (P Q : PROP) (Φ : A → PROP) :
  P ∗ (∃ a, Φ a) ∗ Q ⊢ Q ∗ ∃ a, P ∗ Φ a.
Proof.
  iIntros "[H1 [H2 H3]]".
  iDestruct "H2" as (x) "H2".
  iSplitL "H3".
  – iAssumption.
  – iExists x.
    iFrame.
```

Lemma universally quantified in the BI logic

Making Iris Proof Mode parametric in the separation logic (2)

A **Bunched Implications (BI) logic** [O'Hearn&Pym,99] is a preorder $(\mathsf{Prop}, \vdash)$ with:

- Operations $\mathsf{True}, \mathsf{False}, \wedge, \vee, \Rightarrow, \forall, \exists$ satisfying the axioms of intuitionistic logic
- Operations $\mathsf{emp}, *, -\!\!*$ satisfying:

$$\begin{aligned} \mathsf{emp} * P &\dashv\vdash P \\ P * Q &\vdash Q * P \\ (P * Q) * R &\vdash P * (Q * R) \end{aligned} \qquad \frac{P_1 \vdash Q_1 \qquad P_2 \vdash Q_2}{P_1 * P_2 \vdash Q_1 * Q_2} \qquad \frac{P * Q \vdash R}{\overline{P \vdash Q -\!\!* R}}$$

Making Iris Proof Mode parametric in the separation logic (2)

A **Bunched Implications (BI) logic** [O'Hearn&Pym,99] is a preorder $(\mathsf{Prop}, \vdash)$ with:

- Operations $\mathsf{True}, \mathsf{False}, \land, \lor, \Rightarrow, \forall, \exists$ satisfying the axioms of intuitionistic logic
- Operations $\mathsf{emp}, *, -\!\!*$ satisfying:

$$\begin{array}{rcl} \mathsf{emp} * P & \dashv\vdash & P \\ P * Q & \vdash & Q * P \\ (P * Q) * R & \vdash & P * (Q * R) \end{array} \qquad \frac{P_1 \vdash Q_1 \qquad P_2 \vdash Q_2}{P_1 * P_2 \vdash Q_1 * Q_2} \qquad \frac{P * Q \vdash R}{P \vdash Q -\!\!* R}$$

```
Structure bi := Bi {
  bi_car      :> Type;
  bi_entails : bi_car → bi_car → Prop;
  bi_forall  : ∀ A, (A → bi_car) → bi_car;
  bi_sep      : bi_car → bi_car → bi_car;
  (* other separation logic operators and axioms *)
}.
```

Conclusion

- Separation logic is a good fit for verification of programs with pointers and concurrency
- Separation logic is a good fit for verification of fancy type systems
- There is a lot of fun math in the meta theory of separation logic (categorical models based on step-indexing, modalities, monoids)
- Separation logic is an active research area
- But most importantly: **it is lots of fun!**