Mechanized verification of type systems using Iris

Robbert Krebbers

Radboud University Nijmegen, The Netherlands

January 31, 2024 @ Dagstuhl, Germany

The old problem of proving "type safety":

"Well-typed programs cannot go wrong"

The old problem of proving "type safety":

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

The old problem of proving "type safety":

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

Goal of this talk:

- Introduce the "logical approach in Iris" as an alternative to the standard progress/preservation approach to type safety
- Show that this approach makes it possible to type "unsafe" code
- Show that this approach is well-suited for mechanization of challenging type systems in the Coq proof assistant

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

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

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

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

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

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

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

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

1. **Progress:** If $\vdash e : A$ then $e \in \mathsf{Val}$ or $\mathsf{reducible}(e)$
2. **Preservation:** If $\vdash e : A$ and $e \rightarrow e'$ 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 $e \rightarrow^* e'$ 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 $\Sigma \vdash e : A$ and $\Sigma \vdash_{\mathrm{h}} \sigma$ and $(\sigma, e) \rightarrow (\sigma, e')$ then there exists $\Sigma' \supseteq \Sigma$ such that $\Sigma' \vdash e' : A$ 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 $\Sigma \vdash e : A$ and $\Sigma \vdash_{\mathrm{h}} \sigma$ and $(\sigma, e) \to (\sigma, e')$ then there exists $\Sigma' \supseteq \Sigma$ such that $\Sigma' \vdash e' : A$ and $\Sigma' \vdash_{\mathrm{h}} \sigma'$

- Even more tricky once you consider a substructural type system
  Disjointness conditions show up everywhere
  (And Coq 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 $\Sigma \vdash e : A$ and $\Sigma \vdash_{\mathrm{h}} \sigma$ and $(\sigma, e) \to (\sigma, e')$ then there exists $\Sigma' \supseteq \Sigma$ such that $\Sigma' \vdash e' : A$ and $\Sigma' \vdash_{\mathrm{h}} \sigma'$

- Even more tricky once you consider a substructural type system
  Disjointness conditions show up everywhere
  (And Coq 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 $\vDash e : A$ then $\mathrm{safe}(e)$
2. **Fundamental theorem:** If $\vdash e : A$ then $\vDash e : A$

Semantic typing

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

1. **Adequacy:** If $\vDash e : A$ then safe(e)

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

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

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

Semantic typing

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

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

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

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

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

Semantic typing for STLC

Semantic interpretation of types ("logical relation"):

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

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

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"):

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

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

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

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

Weakest precondition:

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

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

Semantic typing 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:

$$\begin{aligned}
\mathsf{wp}\ _\ \{_\} &: \mathsf{Expr} \to (\mathsf{Val} \to \mathsf{Prop}) \to \mathsf{Prop} \\
\mathsf{wp}\ e\,\{\Phi\} &\triangleq \mathsf{safe}(e) \wedge (\forall v.\, e \to^* v \Rightarrow \Phi\ v)
\end{aligned}$$

Semantic typing judgment:

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

Semantic typing for STLC

Semantic interpretation of types ("logical relation"):

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

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

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

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

Weakest precondition:

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

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

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

Proofs of key properties

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

Proofs of key properties

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

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

Example: Proof of the semantic typing rule for application

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

Proof of the fundamental theorem

Reasoning about the operational semantics is encapsulated by the WP rules

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

Example: Proof of the semantic typing rule for application

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

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

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

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

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

Proof of the fundamental theorem

Reasoning about the operational semantics is encapsulated by the WP rules

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

Example: Proof of the semantic typing rule for application

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

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?

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

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?

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

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

An "unsafe" fixpoint combinator

Consider a strict version of Curry's fixpoint operator:

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

Do we have?

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

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

Do we have?

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

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?

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

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

Do we have?

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

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

Towards "logical typing"

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

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

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

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

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

Towards "logical typing"

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

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

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

Important observations:

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

Towards "logical typing"

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

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

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

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

$$\llbracket A \to B \rrbracket \triangleq \lambda v.\, \forall w.\, \llbracket A \rrbracket\; w \Rightarrow \mathsf{wp}\, (v\;\; w)\, \{\llbracket B \rrbracket\}$$

Important observations:

- Each type is given a semantic interpretation following the Curry-Howard correspondence
- Instead of Coq's Prop we can use **a logic with more fancy connectives** to interpret more challenging types

Towards "logical typing"

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

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

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

Important observations:

- Each type is given a semantic interpretation following the Curry-Howard correspondence
- Instead of Coq's Prop we can use **a logic with more fancy connectives** to interpret more challenging types

separation logic

Towards "logical typing"

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

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

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

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

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

Important observations:

- Each type is given a semantic interpretation following the Curry-Howard correspondence
- Instead of Coq's Prop we can use **a logic with more fancy connectives** to interpret more challenging types

concurrent separation logic

Towards "logical typing"

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

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

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

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

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

Important observations:

- Each type is given a semantic interpretation following the Curry-Howard correspondence
- Instead of Coq's Prop we can use **a logic with more fancy connectives** to interpret more challenging types

higher-order concurrent separation logic

Towards “logical typing”

Recall the semantic interpretation of types (“logical relation”):

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

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

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

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

Important observations:

- Each type is given a semantic interpretation following the Curry-Howard correspondence
- Instead of Coq’s Prop we can use **a logic with more fancy connectives** to interpret more challenging types

Iris

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:

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

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

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

Slightly less basic example:

$$\text{isList}\ \ell\ \vec{v} \triangleq \begin{cases} \ell \mapsto \text{nil} & \text{if } \vec{v} = [\,] \\ \exists \ell'.\, \ell \mapsto \text{cons}\ v_1\ \ell' * \text{isList}\ \ell'\ \vec{v}_2 & \text{if } \vec{v} = v_1 :: \vec{v}_2 \end{cases}$$

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

Slightly less basic example:

$$\text{isList } \ell \ \vec{v} \triangleq \begin{cases} \ell \mapsto \text{nil} & \text{if } \vec{v} = [\,] \\ \exists \ell'.\, \ell \mapsto \text{cons } v_1 \ \ell' * \text{isList } \ell' \ \vec{v}_2 & \text{if } \vec{v} = v_1 :: \vec{v}_2 \end{cases}$$

$$\{\text{isList } \ell_1 \ \vec{v}_1 * \text{isList } \ell_2 \ \vec{v}_2\}\, \mathit{append}\ \ell_1 \ \ell_2 \{\text{isList } \ell_1 \ (\vec{v}_1 \mathbin{++} \vec{v}_2)\}$$

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

Slightly less basic example:

$$\text{isList}\ \ell\ \vec{v} \triangleq \begin{cases} \ell \mapsto \text{nil} & \text{if } \vec{v} = [\,] \\ \exists \ell'.\, \ell \mapsto \text{cons}\ v_1\ \ell' * \text{isList}\ \ell'\ \vec{v}_2 & \text{if } \vec{v} = v_1 :: \vec{v}_2 \end{cases}$$

$$\{\text{isList}\ \ell_1\ \vec{v}_1 * \text{isList}\ \ell_2\ \vec{v}_2\}\, \mathit{append}\ \ell_1\ \ell_2 \{\text{isList}\ \ell_1\ (\vec{v}_1 +\!\!+\ \vec{v}_2)\}$$

the $*$ ensures that all nodes of ℓ_1 and ℓ_2 are disjoint

The simple model of separation logic

The semantic domains:

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

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

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

The simple model of separation logic

The semantic domains:

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

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

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

Entailment:

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

The simple model of separation logic

The semantic domains:

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

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

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

Entailment:

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

The connectives of separation logic:

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

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

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

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

The simple model of separation logic

The semantic domains:

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

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

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

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 ("logical relation"):

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

Semantic typing for a substructural type system

Semantic interpretation of types ("logical relation"):

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

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

Semantic typing for a substructural type system

Semantic interpretation of types ("logical relation"):

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

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

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

Semantic typing for a substructural type system

Semantic interpretation of types ("logical relation"):

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

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

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

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

Semantic typing for a substructural type system

Semantic interpretation of types ("logical relation"):

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

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

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

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

Weakest precondition of separation logic:

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

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

(Modulo "frame baking")

Semantic typing for a substructural type system

Semantic interpretation of types ("logical relation"):

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

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

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

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

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

Semantic typing for a substructural type system

Semantic interpretation of types ("logical relation"):

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

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

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

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

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

Semantic typing for a substructural type system

Semantic interpretation of types ("logical relation"):

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

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

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

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

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

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

Semantic typing for a substructural type system

Semantic interpretation of types ("logical relation"):

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

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

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

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

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

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

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

Semantic typing for a substructural type system

Semantic interpretation of types ("logical relation"):

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

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

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

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

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

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

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

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):

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

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):

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

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

$$\begin{aligned}\texttt{recv}\ c = {} & \texttt{match}\ !c\ \texttt{with} \\ & \quad \texttt{None} \Rightarrow \texttt{recv}\ c \\ & \mid \texttt{Some}\ v \Rightarrow \texttt{free}(c);\ v \\ & \texttt{end}\end{aligned}$$

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):

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

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

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):

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

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

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

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:

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

$$\vDash \texttt{send} : !A \multimap A \multimap ()$$

$$\vDash \texttt{recv} : ?A \multimap A$$

The separation logic API:

$$\{\mathsf{True}\}\ \texttt{new}\ ()\ \{(c_1, c_2).\ \mathsf{IsChan}(c_1, \mathsf{Send}, \Phi) * \mathsf{IsChan}(c_2, \mathsf{Recv}, \Phi)\}$$

$$\{\mathsf{IsChan}(c, \mathsf{Send}, \Phi) * \Phi\ v\}\ \texttt{send}\ c\ v\ \{\mathsf{True}\}$$

$$\{\mathsf{IsChan}(c, \mathsf{Recv}, \Phi)\}\ \texttt{recv}\ c\ \{w.\ \Phi\ w\}$$

Separation logic API for one-shot channels

Recall the desired typing rules:

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

$$\vDash \texttt{send} : !A \multimap A \multimap ()$$

$$\vDash \texttt{recv} : ?A \multimap A$$

The separation logic API:

$$\{\mathsf{True}\}\ \texttt{new}\ ()\ \{(c_1, c_2).\ \mathsf{IsChan}(c_1, \mathsf{Send}, \Phi) * \mathsf{IsChan}(c_2, \mathsf{Recv}, \Phi)\}$$

$$\{\mathsf{IsChan}(c, \mathsf{Send}, \Phi) * \Phi\ v\}\ \texttt{send}\ c\ v\ \{\mathsf{True}\}$$

$$\{\mathsf{IsChan}(c, \mathsf{Recv}, \Phi)\}\ \texttt{recv}\ c\ \{w.\ \Phi\ w\}$$

$\Phi : \mathsf{Val} \to \mathsf{iProp}$, so we can transfer resources

Logical typing for one-shot channels

Semantic interpretation of types ("logical relation"):

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

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

$$[\![?A]\!] \triangleq \lambda c.\, \mathsf{IsChan}(c, \mathsf{Recv}, [\![A]\!])$$

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

Verification of one-shot channel separation logic API

One-shot channel ownership defined using standard Iris methodology

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

Verification of one-shot channel separation logic API

One-shot channel ownership defined using standard Iris methodology:

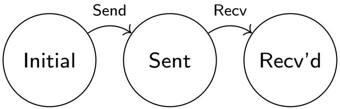
1. Model behavior as a state transition system (STS)

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

Verification of one-shot channel separation logic API

One-shot channel ownership defined using standard Iris methodology:

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

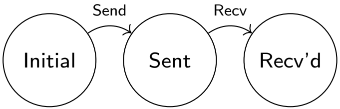


$$\text{IsChan}(c, \mathit{tag}, \Phi) \triangleq \ldots$$

Verification of one-shot channel separation logic API

One-shot channel ownership defined using standard Iris methodology:

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

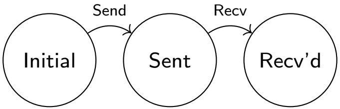


$$\mathsf{IsChan}(c, tag, \Phi) \triangleq \ldots$$

Verification of one-shot channel separation logic API

One-shot channel ownership defined using standard Iris methodology:

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



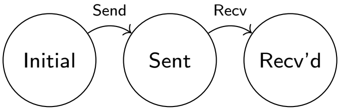
$$\text{chan\_inv} \triangleq (\underbrace{\qquad\qquad}_{\text{(1) initial state}}) \vee (\underbrace{\qquad\qquad\qquad\qquad\qquad}_{\text{(2) message sent, but not yet received}}) \vee (\underbrace{\qquad\qquad\qquad}_{\text{(3) final state}})$$

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

Verification of one-shot channel separation logic API

One-shot channel ownership defined using standard Iris methodology:

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



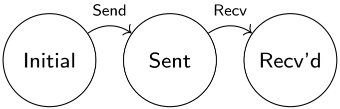
$$\text{chan\_inv} \triangleq (\underbrace{\quad\quad\quad}_{\text{(1) initial state}}) \lor (\underbrace{\quad\quad\quad\quad\quad\quad}_{\text{(2) message sent, but not yet received}}) \lor (\underbrace{\quad\quad\quad}_{\text{(3) final state}})$$

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

Verification of one-shot channel separation logic API

One-shot channel ownership defined using standard Iris methodology:

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



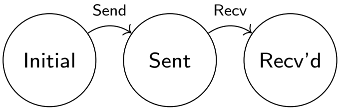
$$\text{chan\_inv} \quad c \quad \triangleq \underbrace{(\ c \mapsto \texttt{None}\ )}_{\text{(1) initial state}} \vee \underbrace{(\exists v.\, c \mapsto \texttt{Some}\, v \qquad)}_{\text{(2) message sent, but not yet received}} \vee \underbrace{(\qquad\qquad)}_{\text{(3) final state}}$$

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

Verification of one-shot channel separation logic API

One-shot channel ownership defined using standard Iris methodology:

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



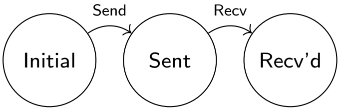
$$\text{chan\_inv} \qquad c\ \Phi \triangleq (\underbrace{c \mapsto \texttt{None}}_{\text{(1) initial state}}) \vee (\underbrace{\exists v.\, c \mapsto \texttt{Some}\, v * \Phi\ v}_{\text{(2) message sent, but not yet received}}) \vee (\underbrace{\qquad\qquad}_{\text{(3) final state}})$$

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

Verification of one-shot channel separation logic API

One-shot channel ownership defined using standard Iris methodology:

1. Model behavior 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



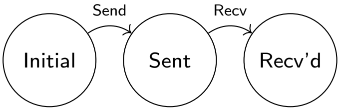
$$\text{chan\_inv} \quad c\ \Phi \triangleq (\underbrace{c \mapsto \texttt{None}}_{\text{(1) initial state}}) \lor (\underbrace{\exists v.\, c \mapsto \texttt{Some}\, v * \Phi\ v}_{\text{(2) message sent, but not yet received}}) \lor (\underbrace{\qquad\qquad}_{\text{(3) final state}})$$

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

Verification of one-shot channel separation logic API

One-shot channel ownership defined using standard Iris methodology:

1. Model behavior 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



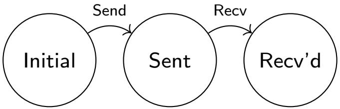
$$\text{chan\_inv}\ \gamma_s \quad c\ \Phi \triangleq (\underbrace{c \mapsto \texttt{None}}_{\text{(1) initial state}}) \lor (\underbrace{\exists v.\, c \mapsto \texttt{Some}\, v * \Phi\ v * \text{tok}\,\gamma_s}_{\text{(2) message sent, but not yet received}}) \lor (\underbrace{\qquad\qquad}_{\text{(3)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

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

Verification of one-shot channel separation logic API

One-shot channel ownership defined using standard Iris methodology:

1. Model behavior 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



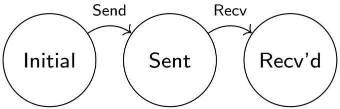
$$\text{chan\_inv}\ \gamma_s\ \gamma_r\ c\ \Phi \triangleq (\underbrace{c \mapsto \texttt{None}}_{\text{(1) initial state}}) \vee (\underbrace{\exists v.\, c \mapsto \texttt{Some}\ v * \Phi\ v * \text{tok}\ \gamma_s}_{\text{(2) message sent, but not yet received}}) \vee (\underbrace{\text{tok}\ \gamma_s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

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

Verification of one-shot channel separation logic API

One-shot channel ownership defined using standard Iris methodology:

1. Model behavior 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
5. Give concurrent actors access to the invariant and their respective ghost state



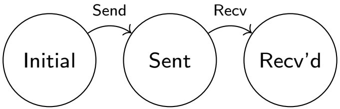
$$\text{chan\_inv}\ \gamma_s\ \gamma_r\ c\ \Phi \triangleq (\underbrace{c \mapsto \texttt{None}}_{\text{(1) initial state}}) \lor (\underbrace{\exists v.\, c \mapsto \texttt{Some}\, v * \Phi\ v * \text{tok}\,\gamma_s}_{\text{(2) message sent, but not yet received}}) \lor (\underbrace{\text{tok}\,\gamma_s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

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

Verification of one-shot channel separation logic API

One-shot channel ownership defined using standard Iris methodology:

1. Model behavior 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
5. Give concurrent actors access to the invariant and their respective ghost state



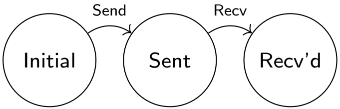
$$\mathsf{chan\_inv}\ \gamma_s\ \gamma_r\ c\ \Phi \triangleq (\underbrace{c \mapsto \texttt{None}}_{\text{(1) initial state}}) \vee (\underbrace{\exists v.\, c \mapsto \texttt{Some}\ v * \Phi\ v * \mathsf{tok}\,\gamma_s}_{\text{(2) message sent, but not yet received}}) \vee (\underbrace{\mathsf{tok}\,\gamma_s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

$$\mathsf{IsChan}(c, tag, \Phi) \triangleq \exists \gamma_s, \gamma_r.\ \boxed{\mathsf{chan\_inv}\ \gamma_s\ \gamma_r\ c\ \Phi} \ldots$$

Verification of one-shot channel separation logic API

One-shot channel ownership defined using standard Iris methodology:

1. Model behavior 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
5. Give concurrent actors access to the invariant and their respective ghost state



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

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

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

Simuliris Compass

Iris-Tini RustHornBelt

Cosmo CQS SeLoC

iGPS Hazel gDOT GoJournal

OCPL Islaris Actris Iron iRC11

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

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?

How to do mechanized 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 mechanized 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 step-indexing

How to do mechanized 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 step-indexing
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 in a proof assistant

How to do mechanized 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 step-indexing
2. **Use the laws of separation logic**: associativity/commutativity of $*$, distributivity of $\exists$ over $*$, $\ldots$
   - Too low-level, already small proofs require many steps
   - Also rather slow in a proof assistant
3. **Use Iris Proof Mode**
   - Topic of today's talk

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

Embedding of a proof assistant for separation logic in Coq

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

Why use Coq and not build a standalone proof assistant for separation logic?

Prove soundness of embedded proof assistant

Reuse infrastructure of host proof assistant

Users do not need to learn new tool

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.
Qed.
```

The hypotheses for the left conjunct

```
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 Coq proofs**
  For many Coq tactics `tac`, we have a variant `iTac`

Features of the Iris Proof Mode

- **Proofs have the look and feel of ordinary Coq proofs**
  For many Coq 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 Coq proofs**
  For many Coq 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 Coq proofs**
  For many Coq 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 Coq's Ltac like ordinary Coq tactics

Changes since the POPL'17 paper on Iris Proof Mode

- Generalized to any Bunched Implications (BI) logic (Krebbers *et al.*, ICFP'18)
- Many usability improvements:
  Smarter tactics, better error messages, improved robustness and performance
- Proof automation: **RefinedC** (Sammler *et al.* PLDI'21), **Diaframe** (Mulder *et al.* PLDI'22, PLDI'23, OOPSLA'23), BedRock Systems (proprietary)

Changes since the POPL'17 paper on Iris Proof Mode

- Generalized to any Bunched Implications (BI) logic (Krebbers *et al.*, ICFP'18)
- Many usability improvements:
  Smarter tactics, better error messages, improved robustness and performance
- Proof automation: **RefinedC** (Sammler *et al.* PLDI'21), **Diaframe** (Mulder *et al.* PLDI'22, PLDI'23, OOPSLA'23), BedRock Systems (proprietary)

Most importantly: Iris (Proof Mode) got users:

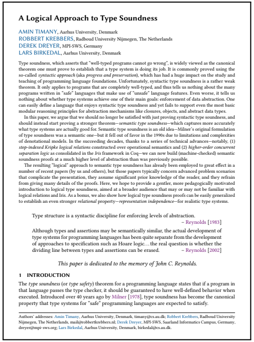
- Coq became essential to teach Iris / concurrent separation logic
- 13 PhD theses
- 98 publications
- 3 editions of the Iris workshop
- Used by researchers at companies: BedRock Systems, Meta, Jetbrains

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

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]

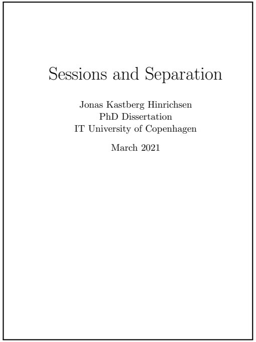
This paper is dedicated to the memory of John C. Reynolds.

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.

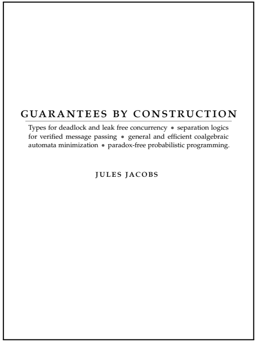
Session types:


Sessions and Separation

Jonas Kastberg Hinrichsen
PhD Dissertation
IT University of Copenhagen

March 2021

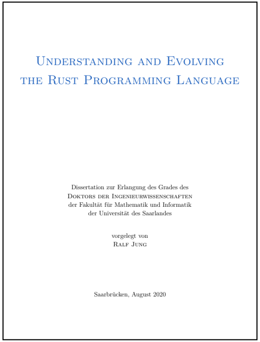
Deadlock freedom:


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/