Reasoning about Programs in Higher-Order Concurrent Separation Logic

Amin Timany

Aarhus University,
Aarhus, Denmark

Mon 18 January 2021
@PLMW'21 – Copenhagen, Denmark (Virtual)

Introduction

As PL researchers we study programs and PL's

An important part of this is proving our programs and PL's are correct

We use mathematical tools:

- Define the semantics of programs, *e.g.*, operational semantics
- State theorems about programs and PL's in semantics terms, *e.g.*, safety, functional correctness, type safety, *etc.*
- Prove these properties using different tools and techniques

Program logics are important tools

- Provide a formal framework for stating and proving properties of programs
- In this talk: the Iris program logic

Some Interesting Properties

one step of computation
(small-step operational semantics)

Safety: program does not crash

$$\text{Safe}(e) \triangleq \forall e'.\ e \underbrace{\longrightarrow^*}_{\text{zero or more steps}} e' \Rightarrow \mathit{Val}(e') \lor \exists e''.\ e' \longrightarrow e''$$

zero or more steps

— Example: Safe($\texttt{letrec}\ f\ x = f\ x\ \texttt{in}\ f\ 4$)

— Counterexample: ¬Safe($\texttt{if}\ \text{“}a\text{”}\ \texttt{then}\ 2\ \texttt{else}\ 3$)

Some Interesting Properties

one step of computation
(small-step operational semantics)

Safety: program does not crash

$$\mathrm{Safe}(e) \triangleq \forall e'.\ e \longrightarrow^* e' \Rightarrow \mathit{Val}(e') \lor \exists e''.\ e' \longrightarrow e''$$

zero or more steps

— Example: $\mathrm{Safe}(\texttt{letrec}\ f\ x = f\ x\ \texttt{in}\ f\ 4)$

— Counterexample: $\neg\mathrm{Safe}(\texttt{if}\ \text{“}a\text{”}\ \texttt{then}\ 2\ \texttt{else}\ 3)$

Functional Correctness: safe, and upon termination postcondition holds

$$\mathrm{Correct}_\phi(e) \triangleq \mathrm{Safe}(e) \land \forall v.\ \mathit{Val}(v) \land e \longrightarrow^* v \Rightarrow \phi(v)$$

— Example: $\mathrm{Correct}_{\mathit{isEven}}(3 + 5)$

Some Interesting Properties

one step of computation
(small-step operational semantics)

Safety: program does not crash

$$\text{Safe}(e) \triangleq \forall e'.\ e \longrightarrow^{*} e' \Rightarrow \mathit{Val}(e') \lor \exists e''.\ e' \longrightarrow e''$$

zero or more steps

— Example: Safe($\texttt{letrec}\ f\ x = f\ x\ \texttt{in}\ f\ 4$)

— Counterexample: $\neg$Safe($\texttt{if}$ "a" $\texttt{then}\ 2\ \texttt{else}\ 3$)

Functional Correctness: safe, and upon termination postcondition holds

$$\text{Correct}_{\phi}(e) \triangleq \text{Safe}(e) \land \forall v.\ \mathit{Val}(v) \land e \longrightarrow^{*} v \Rightarrow \phi(v)$$

— Example: $\text{Correct}_{\mathit{isEven}}(3 + 5)$

Type safety: well-typed programs are safe

Some Interesting Properties

one step of computation
(small-step operational semantics)

Safety: program does not crash

$$\text{Safe}(e) \triangleq \forall e'.\ e \rightarrow^* e' \Rightarrow \mathit{Val}(e') \lor \exists e''.\ e' \rightarrow e''$$

zero or more steps

— Example: $\text{Safe}(\texttt{letrec}\ f\ x = f\ x\ \texttt{in}\ f\ 4)$

— Counterexample: $\neg\text{Safe}(\texttt{if}\ \text{“}a\text{”}\ \texttt{then}\ 2\ \texttt{else}\ 3)$

Functional Correctness: safe, and upon termination postcondition holds

$$\text{Correct}_\phi(e) \triangleq \text{Safe}(e) \land \forall v.\ \mathit{Val}(v) \land e \rightarrow^* v \Rightarrow \phi(v)$$

— Example: $\text{Correct}_{\mathit{isEven}}(3 + 5)$

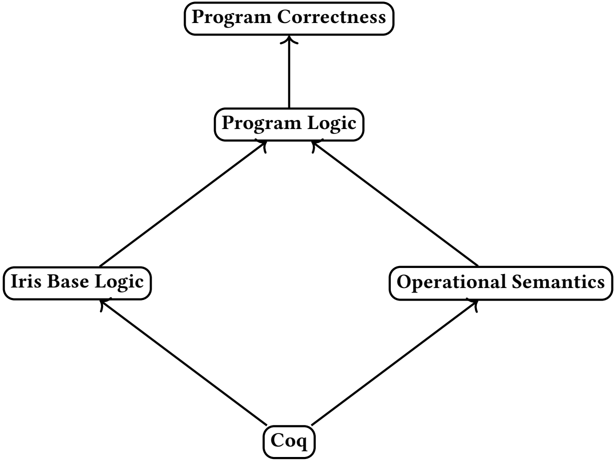
Type safety: well-typed programs are safe

In this talk: Iris and how it helps us prove such properties

What is Iris?

A Framework for Higher-order Concurrent Separation Logics

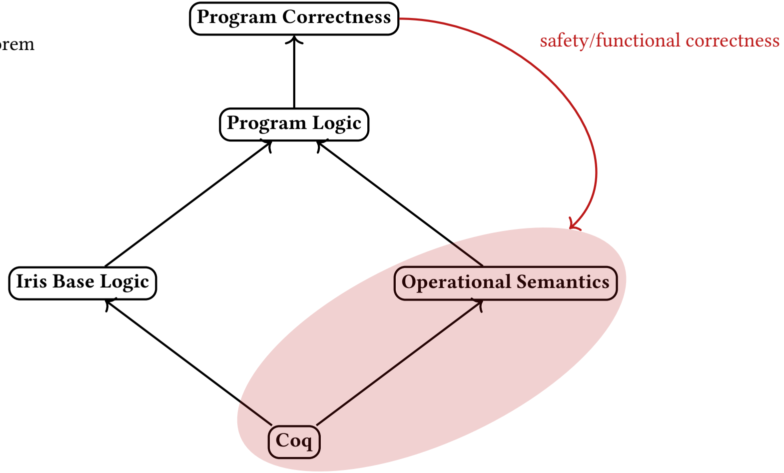
$\longrightarrow$: Built on top of



What is Iris?

A Framework for Higher-order Concurrent Separation Logics

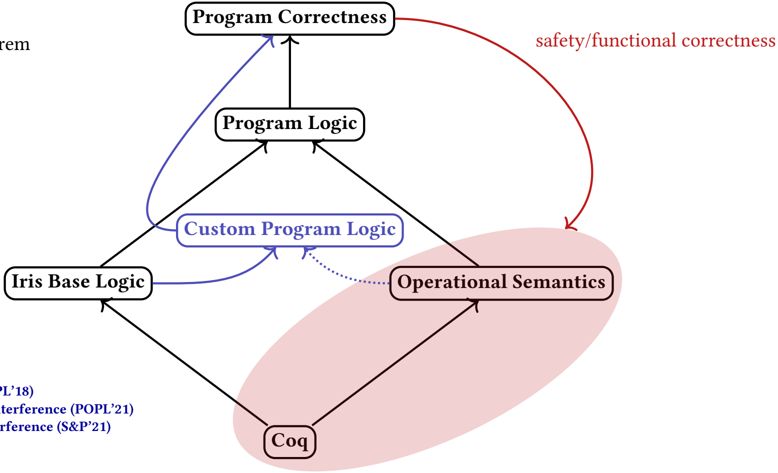
$\rightarrow$: Built on top of
$\rightarrow$: Iris's adequacy theorem



What is Iris?

A Framework for Higher-order Concurrent Separation Logics

→ : Built on top of
→ : Iris's adequacy theorem
□ : User-defined



Used *e.g.* in reasoning about:
– a Haskell-style ST monad (POPL'18)
– termination insensitive non-interference (POPL'21)
– termination sensitive non-interference (S&P'21)

Versatility of Iris

Iris have been used in many projects:

- Reasoning about session types (@CPP'21)
- Reasoning about capability machines (hardware language) (@POPL'21)
- Reasoning about non-interference (a security property) (@POPL'21)
- Reasoning about distributed systems (@POPL'21)
- Proving properties of gradual typing systems (@POPL'21)
- Reasoning about algebraic effect handlers (@POPL'21)
- Reasoning principles for weak memory
- Proving properties of DOT (core of Scala)
- Proving properties of the Rust programming language
- *etc.*

This versatility is due to Iris's expressivity.

Iris Base Logic

A logic with features designed for defining program logics:

$$
\begin{aligned}
P ::= {} & \mathsf{True} \mid \mathsf{False} \mid P \lor P \mid P \land P \mid P \to P \mid \forall x.\ P \mid \exists x.\ P \mid && \text{(higher-order logic)} \\
& P * P \mid && \text{(separation logic)} \\
& \boxed{a}^{\gamma} \mid {\mid\!\Rrightarrow} P \mid && \text{(user-defined resources)} \\
& \triangleright P \mid \mu r.P \mid && \text{(step indexing)} \\
& \boxed{P} && \text{(invariants)}
\end{aligned}
$$

Base logic inference rules:

Löb-ind $\dfrac{\triangleright P \vdash P}{\vdash P}$ ▷-intro $P \vdash \triangleright P$ ▷-mono $\dfrac{P \vdash Q}{\triangleright P \vdash \triangleright Q}$ ∗-true $P * \mathsf{True} \dashv\vdash P$ ∗-elim-L $P * Q \vdash P$ ∗-elim-R $P * Q \vdash Q$ ∗-mono $\dfrac{P_1 \vdash P_2 \qquad Q_1 \vdash Q_2}{P_1 * Q_1 \vdash P_2 * Q_2}$ ∗-comm $P * Q \vdash Q * P$ ∗-assoc $(P * Q) * R \vdash P * (Q * R)$ ∧-intro $\dfrac{P \vdash Q \qquad P \vdash R}{P \vdash Q \land R}$ ⊢-refl $P \vdash P$ ⊢-trans $\dfrac{P \vdash Q \qquad Q \vdash R}{P \vdash R}$ ⊢-true $P \vdash \mathsf{True}$ ⊢-falso $\mathsf{False} \vdash P$ ∧-true $P \land \mathsf{True} \dashv\vdash P$

∧-elim-L $P \land Q \vdash P$ ∧-elim-R $P \land Q \vdash Q$ ∧-mono $\dfrac{P_1 \vdash P_2 \qquad Q_1 \vdash Q_2}{P_1 \land Q_1 \vdash P_2 \land Q_2}$ ∧-comm $P \land Q \vdash Q \land P$ ∧-assoc $(P \land Q) \land R \vdash P \land (Q \land R)$ ∨-intro-L $\dfrac{P \vdash Q}{P \vdash Q \lor R}$ ∨-intro-R $\dfrac{P \vdash R}{P \vdash Q \land R}$ ∨-false $P \lor \mathsf{False} \dashv\vdash P$ ∨-elim $\dfrac{P \vdash R \qquad Q \vdash R}{P \lor Q \vdash R}$ ∨-mono $\dfrac{P_1 \vdash P_2 \qquad Q_1 \vdash Q_2}{P_1 \lor Q_1 \vdash P_2 \lor Q_2}$ ∨-comm $P \lor Q \vdash Q \lor P$ ∨-assoc $(P \lor Q) \lor R \vdash P \lor (Q \lor R)$

Program Logic

A Hoare-style logic:

program

binder for return value

$$\{P\}\; e \;\{x.\; Q\}$$

precondition

postcondition

Examples: $\{n \geq 0\}\; \mathit{fact}\ n \;\{x.\; x = n!\}$ $\qquad$ $\{\mathsf{True}\}\; \texttt{letrec}\ f\ x = f\ x\ \texttt{in}\ f\ 4 \;\{x.\; \mathsf{False}\}$

Program Logic

A Hoare-style logic:

program

binder for return value

$$\{P\}\ e\ \{x.\ Q\}$$

precondition

postcondition

Examples: $\{n \geq 0\}\ \mathit{fact}\ n\ \{x.\ x = n!\}$ $\{\mathsf{True}\}\ \texttt{letrec}\ f\ x = f\ x\ \texttt{in}\ f\ 4\ \{x.\ \mathsf{False}\}$

Theorem (Adequacy)

If we prove

$$\vdash \{\mathit{True}\}\ e\ \{x.\ \phi(x)\}$$

in Iris *for a suitable* ϕ*, then* $\mathrm{Correct}_{\phi}(e)$

Program Logic

A Hoare-style logic:

$$\{\underbrace{P}_{\text{precondition}}\}\ \overbrace{e}^{\text{program}}\ \{\overbrace{x}^{\text{binder for return value}}.\ \underbrace{Q}_{\text{postcondition}}\}$$

Examples: $\{n \geq 0\}\ \mathit{fact}\ n\ \{x.\ x = n!\}$ $\{\mathsf{True}\}\ \texttt{letrec}\ f\ x = f\ x\ \texttt{in}\ f\ 4\ \{x.\ \mathsf{False}\}$

Theorem (Adequacy)

If we prove

$$\vdash \{\mathit{True}\}\ e\ \{x.\ \phi(x)\}$$

in Iris *for a suitable* ϕ, *then* $\mathsf{Correct}_\phi(e)$

Proof rules for reasoning about programs:

Hoare-Frame
$$\frac{\{P\}\ e\ \{x.\ Q\}}{\{P * R\}\ e\ \{x.\ Q * R\}}$$

Hoare-Bind
$$\frac{\{P\}\ e\ \{x.\ Q\} \qquad \forall v.\ \{Q[v/x]\}\ K[v]\ \{x.\ R\}}{\{P\}\ K[e]\ \{x.\ R\}}$$

Hoare-Consequence
$$\frac{\{P\}\ e\ \{x.\ Q\} \qquad P' \vdash P \qquad \forall v.\ Q[v/x] \vdash Q'[v/x]}{\{P'\}\ e\ \{x.\ Q'\}}$$

Hoare-rec
$$\frac{\triangleright \{P\}\ e[(\texttt{rec}\ f\ x = e)/f][v/x]\ \{x.\ Q\}}{\{P\}\ (\texttt{rec}\ f\ x = e)\ v\ \{x.\ Q\}}$$

Hoare-if-true
$$\frac{\{P\}\ e_1\ \{x.\ Q\}}{\{P\}\ \texttt{if true then}\ e_1\ \texttt{else}\ e_2\ \{x.\ Q\}}$$

Hoare-if-false
$$\frac{\{P\}\ e_2\ \{x.\ Q\}}{\{P\}\ \texttt{if false then}\ e_1\ \texttt{else}\ e_2\ \{x.\ Q\}}$$

Hoare-alloc
$$\{\mathsf{True}\}\ \texttt{ref}\ v\ \{x.\ \exists \ell.\ x = \ell * \ell \mapsto v\}$$

Hoare-load
$$\{\ell \mapsto v\}\ !\,\ell\ \{x.\ x = v * \ell \mapsto v\}$$

Hoare-store
$$\{\ell \mapsto v\}\ \ell \leftarrow w\ \{x.\ x = () * \ell \mapsto w\}$$

Hoare-faa
$$\{\ell \mapsto n\}\ \texttt{faa}\ \ell\ m\ \{x.\ \ell \mapsto (n + m)\}$$

Hoare-par
$$\frac{\{P_1\}\ e_1\ \{x.\ Q_1\} \qquad \{P_2\}\ e_2\ \{x.\ Q_2\}}{\{P_1 * P_2\}\ e_1 \| e_2\ \{x.\ \exists v_1, v_2.\ x = (v_1, v_2) * Q_1[v_1/x] * Q_2[v_2/x]\}}$$

Hoare-inv-alloc
$$\frac{\{P * \boxed{R}\}\ e\ \{x.\ Q\}}{\{P * R\}\ e\ \{x.\ Q\}}$$

Expressivity: Higher-Order Logic

Specifying abstract data types:[1]

$$
\begin{aligned}
&\exists \mathit{isStack} : \mathit{Val} \to \mathit{list}(\mathit{Val} \to \mathrm{Prop}) \to \mathrm{Prop}.\\
&\qquad \{\mathrm{True}\}\ \mathrm{mk\_stack}()\ \{s.\mathit{isStack}(s, [])\} \land\\
&\qquad \forall s. \forall \Phi. \forall \Phi s. \{\mathit{isStack}(s, \Phi s) * \Phi(x)\}\ \mathrm{push}(x, s)\ \{v.v = () \land \mathit{isStack}(s, \Phi :: \Phi s)\} \land\\
&\qquad \forall s. \forall \Phi. \forall \Phi s. \{\mathit{isStack}(s, \Phi :: \Phi s)\}\ \mathrm{pop}(s)\ \{v.\Phi(v) * \mathit{isStack}(s, \Phi s)\}
\end{aligned}
$$

Note the higher-order quantification of a predicate that takes a list of predicates

[1] Taken verbatim from Iris lecture notes.

Expressivity: Separation Logic

Separating conjunction:

separating conjunction

$$P \ast Q$$

$P \ast Q$ holds if P and Q hold for *disjoint* resources

Example: exclusive ownership of a memory location (points-to proposition)

$$\ell \mapsto v \ast \ell' \mapsto v' \vdash \ell \neq \ell'$$

Hoare-alloc

$$\{\mathsf{True}\}\ \texttt{ref}\ v\ \{x.\ \exists \ell.\ x = \ell \ast \ell \mapsto v\}$$

Hoare-load

$$\{\ell \mapsto v\}\ !\,\ell\ \{x.\ x = v \ast \ell \mapsto v\}$$

Hoare-store

$$\{\ell \mapsto v\}\ \ell \leftarrow w\ \{x.\ x = () \ast \ell \mapsto w\}$$

Expressivity: Separation Logic

In separation logic a Hoare triple specifies *footprint* of the program.

Hence the *frame* rule:

$$\textsc{Hoare-Frame}\quad \frac{\{P\}\, e\, \{x.\ Q\}}{\{P * R\}\, e\, \{x.\ Q * R\}}$$

Important for modular verification:
Verify modules working on separate parts of memory in isolation and combine proofs

What if two modules share memory?
We use invariants (and resources) to specify sharing protocols

Expressivity: User-Defined Resources

Users can introduce resources as partial commutative monoids (PCM's)

logical equivalence user-defined operation

$$\boxed{a}^{\gamma} * \boxed{b}^{\gamma} \dashv\vdash \boxed{a \cdot b}^{\gamma}$$

update modality

$\mathrel{|\!\!\Rrightarrow} P$ **holds if** P **holds after updating resources**

Idea: for verifying stateful programs we need a stateful logic

Expressivity: Step-Indexing and Invariants

Iris invariants are *impredicative*:

P can be any proposition;
it may also including invariants

$\boxed{P}$

Step-indexing is necessary for impredicative invariants to avoid self-referential paradoxes

These features are necessary for defining logical relations models for programming languages with advanced features

Expressivity: Step-Indexing and Invariants (Logical Relations)

Goal: we want to prove type safety (well-typed programs do not crash)

Using logical relations:
We prove by induction on typing derivation:

$$e : \tau \Rightarrow LR_\tau(e)$$

where

$$LR_\tau(e) \Rightarrow \mathsf{Safe}(e)$$

However, we cannot take $LR_\tau(e)$ to be $\mathsf{Safe}(e)$:

$$\mathsf{Safe}(e_1) \wedge \mathsf{Safe}(e_2) \not\Rightarrow \mathsf{Safe}(e_1 - e_2)$$

Counter example: $\mathsf{Safe}(\texttt{true})$ and $\mathsf{Safe}(3)$ but $\neg\mathsf{Safe}(\texttt{true} - 3)$

Expressivity: Step-Indexing and Invariants (Logical Relations)

We should take LR_τ to be:

$$LR_\tau(e) \triangleq \mathsf{Correct}_{[\![\tau]\!]}(e)$$

where $[\![\tau]\!]\,(v)$ means that v is a value of type τ.

Ideally, we should define this by induction on types:

$$
\begin{aligned}
[\![int]\!]\,(v) &\triangleq v \in \mathbb{Z} \\
[\![(\tau_1 \times \tau_2)]\!]\,(v) &\triangleq \exists v1, v_2 . v = (v_1, v_2) \land [\![\tau_1]\!]\,(v_1) \land [\![\tau_2]\!]\,(v_2) \\
&\vdots \\
[\![\mu X.\, \tau]\!]\,(v) &\triangleq \exists w.\ v = \texttt{fold}\,(w) \land [\![\tau]\!]_{X \Rightarrow [\![\mu X.\, \tau]\!]}\,(w) \\
[\![\texttt{ref}(\tau)]\!]\,(v) &\triangleq \exists \ell.\ v = \ell \land \underline{\ell \text{ always stores a value of } \tau}
\end{aligned}
$$

circular definition

how do we express this?

We use Iris and define

$$LR_\tau(e) \triangleq \{\mathsf{True}\}\, e \,\{x.\ [\![\tau]\!]\,(v)\}$$

We define $[\![\tau]\!]\,(v)$ inductively as follows:

$$
\begin{aligned}
[\![int]\!]\,(v) &\triangleq v \in \mathbb{Z} \\
[\![(\tau_1 \times \tau_2)]\!]\,(v) &\triangleq \exists v1, v_2 . v = (v_1, v_2) \land [\![\tau_1]\!]\,(v_1) \land [\![\tau_2]\!]\,(v_2) \\
&\ \ \vdots \\
[\![\mu X.\, \tau]\!] &\triangleq \mu r.\ \lambda v.\ \exists w.\ v = \texttt{fold}\,(w) \land \triangleright [\![\tau]\!]_{X \mapsto r}\,(w) \\
[\![\texttt{ref}(\tau)]\!]\,(v) &\triangleq \exists \ell.\ v = \ell \land \boxed{\exists w.\ \ell \mapsto w * [\![\tau]\!]\,(w)}
\end{aligned}
$$

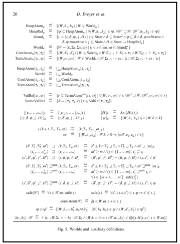
Iris's guarded recursion

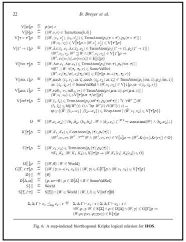
may include invariants

Expressivity: Step-Indexing and Invariants (Logical Relations)

$$
\begin{aligned}
[\![\Xi \vdash \alpha]\!]_\Delta(v, v') &\triangleq \Delta(\alpha)(v, v') \\
[\![\Xi \vdash \mathbb{N}]\!]_\Delta(v, v') &\triangleq v = v' \in \mathbb{N} \\
[\![\Xi \vdash \tau_1 \times \tau_2]\!]_\Delta(v, v') &\triangleq \exists v_1, v_2, v_1', v_2'.\, v = (v_1, v_2) \wedge v' = (v_1', v_2') \wedge \\
&\qquad [\![\Xi \vdash \tau_1]\!]_\Delta(v_1, v_1') \wedge [\![\Xi \vdash \tau_2]\!]_\Delta(v_2, v_2') \\
[\![\Xi \vdash \tau_1 \to \tau_2]\!]_\Delta(v, v') &\triangleq \forall w, w'.\, \Box([\![\Xi \vdash \tau_1]\!]_\Delta(w, w') \mathrel{-\!\!*} \mathcal{E}[\![\Xi \vdash \tau_2]\!]_\Delta(v\ w, v'\ w')) \\
[\![\Xi \vdash \forall \alpha.\, \tau]\!]_\Delta(v, v') &\triangleq \forall f.\, \Box([\![\alpha, \Xi \vdash \tau]\!]_{\Delta[\alpha \mapsto f]}(v\ \_, v'\ \_)) \\
[\![\Xi \vdash \mu\alpha.\, \tau]\!]_\Delta(v, v') &\triangleq \mu f.\, \exists w, w'.\, v = \texttt{fold}\ w \wedge v' = \texttt{fold}\ w' \wedge \\
&\qquad \triangleright [\![\alpha, \Xi \vdash \tau]\!]_{\Delta[\alpha \mapsto f]}(w, w') \\
[\![\Xi \vdash \texttt{ref}(\tau)]\!]_\Delta(v, v') &\triangleq \exists \ell.\, v = \ell \wedge v' = \ell' \wedge \\
&\qquad \boxed{\exists w, w'.\, \ell \mapsto w * \ell' \mapsto w' * [\![\Xi \vdash \tau]\!]_\Delta(w, w')}^{\mathcal{N}.\ell}
\end{aligned}
$$

$$
\mathcal{E}[\![\Xi \vdash \tau]\!]_\Delta(e, e') \triangleq \forall \rho, j, K, \{spec\_inv(\rho) * j \Rrightarrow e'\}\ e\ \{v.\, \exists v'.\, j \Rrightarrow v' * [\![\Xi \vdash \tau]\!]_\Delta(v, v')\}
$$





This approach to type safety is called *semantic type safety*

It has been used for reasoning about correctness of the Rust type system.

See Derek Dreyer's POPL'18 keynote for more details.

Example: Shared Memory Concurrency

Consider the following concurrent program where threads share memory:

$$\texttt{let}\ c = \texttt{ref}\ 0\ \texttt{in}$$
$$(\texttt{faa}\ c\ 1 \parallel \texttt{faa}\ c\ 2)\ ;$$
$$!\,c$$

atomic *fetch and add* operation

Example: Shared Memory Concurrency

Consider the following concurrent program where threads share memory:

$\{\mathsf{True}\}$

$\quad \texttt{let}\ c = \texttt{ref}\ 0\ \texttt{in}$

$\quad (\texttt{faa}\ c\ 1 \parallel \texttt{faa}\ c\ 2)\ ;$

$\quad !\, c$

$\{x.\ x \geq 0\}$

Example: Shared Memory Concurrency

Consider the following concurrent program where threads share memory:

$\{\mathsf{True}\}$

$\quad \texttt{let}\ c = \texttt{ref}\ 0\ \texttt{in}$

$\quad \{c \mapsto 0\}$

$\quad \{\boxed{\exists n.\ n \geq 0 * c \mapsto n}\}$

$$\left(\begin{array}{l} \{\boxed{\exists n.\ n \geq 0 * c \mapsto n}\} \\ \{\exists n.\ n \geq 0 * c \mapsto n\} \\ \quad \texttt{faa}\ c\ 1 \\ \{x.\ \exists n.\ n \geq 0 * c \mapsto n\} \end{array} \middle\| \begin{array}{l} \{\boxed{\exists n.\ n \geq 0 * c \mapsto n}\} \\ \{\exists n.\ n \geq 0 * c \mapsto n\} \\ \quad \texttt{faa}\ \ell\ 2 \\ \{x.\ \exists n.\ n \geq 0 * c \mapsto n\} \end{array}\right);$$

$\quad \{\boxed{\exists n.\ n \geq 0 * c \mapsto n}\}$

$\quad !\,c$

$\{x.\ x \geq 0\}$

Example: Shared Memory Concurrency (Stronger Postcondition)

Can we also prove the following stronger specs for our code?

$$\{\text{True}\}$$

$$\quad \texttt{let}\ c = \texttt{ref}\ 0\ \texttt{in}$$

$$\quad (\texttt{faa}\ c\ 1 \,\|\, \texttt{faa}\ c\ 2)\ ;$$

$$\quad !\,c$$

$$\{x.\ x = 3\}$$

Example: Shared Memory Concurrency (Stronger Postcondition)

Can we also prove the following stronger specs for our code?

With which invariant should we proceed?

$$\boxed{\exists n.\ n \geq 0 * c \mapsto n} \qquad \boxed{c \mapsto 3}$$

Neither works. We need to be able to refer to the value outside the invariant!

$\{\mathsf{True}\}$

$\quad \texttt{let}\ c = \texttt{ref}\ 0\ \texttt{in}$

$\quad (\texttt{faa}\ c\ 1 \,\|\, \texttt{faa}\ c\ 2)\,;$

$\quad !\,c$

$\{x.\ x = 3\}$

Example: Shared Memory Concurrency (Stronger Postcondition)

We use user-defined resources to define the following:

Sum^{γ} — Sum of contributions

Left^{γ} — contributions of the left thread

Right^{γ} — contributions of the right thread

Contr-alloc

$$\vdash \;\Rrightarrow \exists \gamma.\ \mathsf{Sum}^{\gamma}(0) * \mathsf{Left}^{\gamma}(0) * \mathsf{Right}^{\gamma}(0)$$

Contr-sum

$$\mathsf{Sum}^{\gamma}(n) * \mathsf{Left}^{\gamma}(m) * \mathsf{Right}^{\gamma}(k) \vdash n = m + k$$

Contr-incr-left

$$\mathsf{Sum}^{\gamma}(n) * \mathsf{Left}^{\gamma}(m) \vdash \;\Rrightarrow \mathsf{Sum}^{\gamma}(n+k) * \mathsf{Left}^{\gamma}(m+k)$$

Contr-incr-right

$$\mathsf{Sum}^{\gamma}(n) * \mathsf{Right}^{\gamma}(m) \vdash \;\Rrightarrow \mathsf{Sum}^{\gamma}(n+k) * \mathsf{Right}^{\gamma}(m+k)$$

Example: Shared Memory Concurrency (Stronger Postcondition)

Can we also prove the following stronger specs for our code?

$$\{\mathsf{True}\}$$

$$\texttt{let}\ c = \texttt{ref}\ 0\ \texttt{in}$$

$$(\texttt{faa}\ c\ 1 \,\|\, \texttt{faa}\ c\ 2)\,;$$

$$!\,c$$

$$\{x.\ x = 3\}$$

Example: Shared Memory Concurrency (Stronger Postcondition)

Can we also prove the following stronger specs for our code?

$$\{\mathsf{True}\}$$

$$\texttt{let}\ c = \texttt{ref}\ 0\ \texttt{in}$$

$$\{c \mapsto 0\}$$

$$\{c \mapsto 0 * \mathsf{Sum}^{\gamma}(0) * \mathsf{Left}^{\gamma}(0) * \mathsf{Right}^{\gamma}(0)\}$$

$$\{\boxed{\exists n.\ c \mapsto n * \mathsf{Sum}^{\gamma}(n)} * \mathsf{Left}^{\gamma}(0) * \mathsf{Right}^{\gamma}(0)\}$$

$$\left(\begin{array}{l} \{\boxed{\exists n.\ c \mapsto n * \mathsf{Sum}^{\gamma}(n)} * \mathsf{Left}^{\gamma}(0)\} \\ \quad \texttt{faa}\ c\ 1 \\ \{x.\ \mathsf{Left}^{\gamma}(1)\} \end{array} \middle\| \begin{array}{l} \{\boxed{\exists n.\ c \mapsto n * \mathsf{Sum}^{\gamma}(n)} * \mathsf{Right}^{\gamma}(0)\} \\ \quad \texttt{faa}\ c\ 2 \\ \{x.\ \mathsf{Right}^{\gamma}(2)\} \end{array}\right);$$

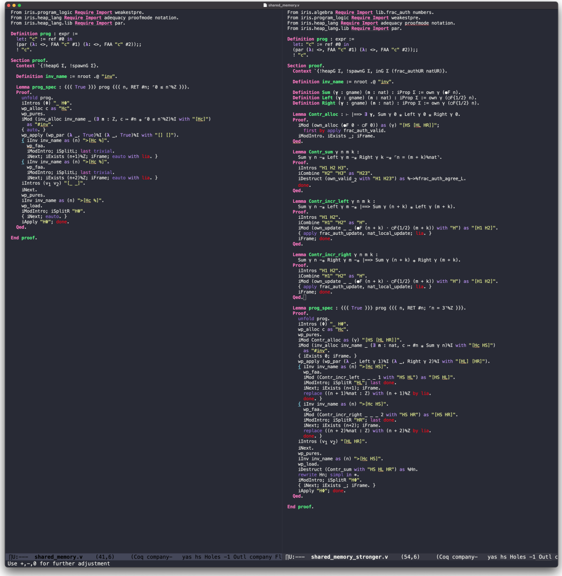
$$\{\boxed{\exists n.\ c \mapsto n * \mathsf{Sum}^{\gamma}(n)} * \mathsf{Left}^{\gamma}(1) * \mathsf{Right}^{\gamma}(2)\}$$

$$!\,c$$

$$\{x.\ x = 3\}$$

Proofs and Iris Proof Mode

- I simplified the proofs that I just presented
- However, Iris features a Proof Mode (IPM)
- IPM makes program verification in Coq very close to what I presented
- To the right: screenshot of the proofs we just saw in Iris in Coq



Online resources

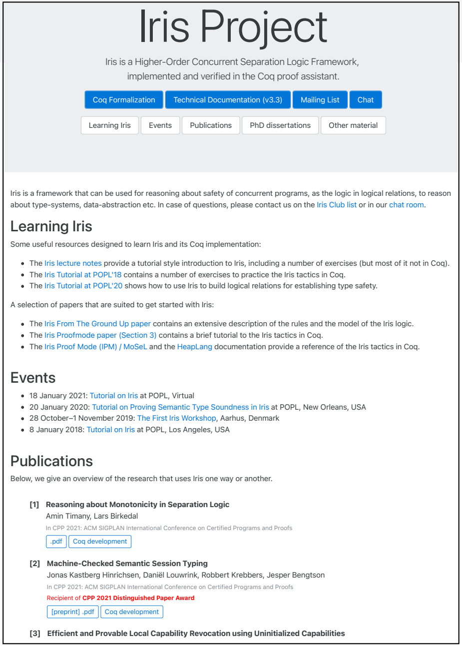
I hope this talk has made you interested in learning more about Iris, separation logic, program verification, *etc.*

See http://iris-project.org

- Iris Tutorial material
- Iris related publications
- PhD theses that include Iris works
- Other manuscripts

See https://cs.au.dk/~timany/talks/plmw21/

- These slides
- A list of Iris related talks at POPL'21


Iris Project

Iris is a Higher-Order Concurrent Separation Logic Framework, implemented and verified in the Coq proof assistant.

Coq Formalization | Technical Documentation (v3.3) | Mailing List | Chat

Learning Iris | Events | Publications | PhD dissertations | Other material

Iris is a framework that can be used for reasoning about safety of concurrent programs, as the logic in logical relations, to reason about type-systems, data-abstraction etc. In case of questions, please contact us on the Iris Club list or in our chat room.

Learning Iris

Some useful resources designed to learn Iris and its Coq implementation:

- The Iris lecture notes provide a tutorial style introduction to Iris, including a number of exercises (but most of it not in Coq).
- The Iris Tutorial at POPL'18 contains a number of exercises to practice the Iris tactics in Coq.
- The Iris Tutorial at POPL'20 shows how to use Iris to build logical relations for establishing type safety.

A selection of papers that are suited to get started with Iris:

- The Iris From The Ground Up paper contains an extensive description of the rules and the model of the Iris logic.
- The Iris Proofmode paper (Section 3) contains a brief tutorial to the Iris tactics in Coq.
- The Iris Proof Mode (IPM) / MoSeL and the HeapLang documentation provide a reference of the Iris tactics in Coq.

Events

- 18 January 2021: Tutorial on Iris at POPL, Virtual
- 20 January 2020: Tutorial on Proving Semantic Type Soundness in Iris at POPL, New Orleans, USA
- 28 October–1 November 2019: The First Iris Workshop, Aarhus, Denmark
- 8 January 2018: Tutorial on Iris at POPL, Los Angeles, USA

Publications

Below, we give an overview of the research that uses Iris one way or another.

[1] Reasoning about Monotonicity in Separation Logic
Amin Timany, Lars Birkedal
In CPP 2021: ACM SIGPLAN International Conference on Certified Programs and Proofs
.pdf | Coq development

[2] Machine-Checked Semantic Session Typing
Jonas Kastberg Hinrichsen, Daniël Louwrink, Robbert Krebbers, Jesper Bengtson
In CPP 2021: ACM SIGPLAN International Conference on Certified Programs and Proofs
Recipient of CPP 2021 Distinguished Paper Award
[preprint] .pdf | Coq development

[3] Efficient and Provable Local Capability Revocation using Uninitialized Capabilities

Iris related talks at POPL'21 and co-events (Time in CET)

Toward Complete Stack Safety for Capability Machines — @PriSC, on 17^{th} at 20:12
Aïna Linn Georges, Armaël Guéneau, Alix Trieu, Lars Birkedal

Contextual Refinement of the Micheal-Scott Queue (Proof Pearl) — @CPP, on 18^{th} at 17:00
Amin Timany, Lars Birkedal

Reasoning About Monotonicity in Separation Logic — @CPP, on 18^{th} at 17:15
Amin Timany, Lars Birkedal

[T4] Iris – A Modular Foundation for Higher-Order Concurrent Separation Logic — @Tutorial, on 18^{th} at 18:00
Tej Chajed, Ralf Jung, Joseph Tassarotti

Machine-Checked Semantic Session Typing — @CPP, on 18^{th} at 18:30
Jonas Kastberg Hinrichsen, Daniël Louwrink, Robbert Krebbers, Jesper Bengtson

Fully Abstract from Static to Gradual — @POPL, on 20^{th} at 16:00
Koen Jacobs, Amin Timany, Dominique Devriese

Efficient and Provable Local Capability Revocation using Uninitialized Capabilities — @POPL, on 21^{st} at 16:00
Aïna Linn Georges, Armaël Guéneau, Thomas Van Strydonck, Amin Timany, Alix Trieu, Sander Huyghebaert, Dominique Devriese, Lars Birkedal

Mechanized Logical Relations for Termination-Insensitive Noninterference — @POPL, on 21^{st} at 16:10
Simon Oddershede Gregersen, Johan Bay, Amin Timany, Lars Birkedal

A Separation Logic for Effect Handlers — @POPL, on 22^{nd} at 16:15
Paulo Emílio de Vilhena, François Pottier

Distributed Causal Memory: Modular Specification and Verification in Higher-Order Distributed Separation Logic — @POPL, on 22^{nd} at 16:15
Léon Gondelman, Simon Oddershede Gregersen, Abel Nieto, Amin Timany, Lars Birkedal

Thanks.