Reasoning about Programs in Higher-Order Concurrent Separation Logic

Amin Timany

Aarhus University,
Aarhus, Denmark

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

Safety: program does not crash

\[ \text{Safe}(e) \triangleq \forall e'. \ e \rightarrow^* e' \Rightarrow \text{Val}(e') \vee \exists e''. \ e' \rightarrow e'' \]

- Example: Safe(\texttt{letrec } f \ x = f \ x \ \texttt{in} \ f 4)
- Counterexample: \neg \text{Safe}(\texttt{if "a" then 2 else 3})
Some Interesting Properties

Safety: program does not crash

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

zero or more steps

— Example: Safe(letrec f x = f x in f 4)

— Counterexample: $\lnot$Safe(if "a" then 2 else 3)

Functional Correctness: safe, and upon termination postcondition holds

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

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

Safety: program does not crash

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

zero or more steps

– Example: Safe(letrec \( f \ x = f \ x \ \text{in} \ f \ 4 \))

– Counterexample: \( \neg \text{Safe}(\text{if} \ "a" \ \text{then} \ 2 \ \text{else} \ 3) \)

Functional Correctness: safe, and upon termination postcondition holds

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

– Example: Correct_{\text{isEven}}(3 + 5)

Type safety: well-typed programs are safe
Some Interesting Properties

Safety: program does not crash

\[ \text{Safe}(e) \triangleq \forall e'. \ e \to^* e' \Rightarrow \text{Val}(e') \lor \exists e''. \ e' \to e'' \]

- Example: \text{Safe}(\text{letrec } f x = f x \text{ in } f 4)
- Counterexample: \neg \text{Safe}(\text{if } "a" \text{ then } 2 \text{ else } 3)

Functional Correctness: safe, and upon termination postcondition holds

\[ \text{Correct}_\phi(e) \triangleq \text{Safe}(e) \land \forall v. \ \text{Val}(v) \land e \to^* v \Rightarrow \phi(v) \]

- Example: \text{Correct}_{\text{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

→ : Built on top of
What is Iris?

A Framework for Higher-order Concurrent Separation Logics

→ : Built on top of
→ : Iris’s adequacy theorem

Program Correctness

Program Logic

Iris Base Logic

Operational Semantics

Coq

safety/functional correctness
What is Iris?

A Framework for Higher-order Concurrent Separation Logics

→ : Built on top of
→ : Iris’s adequacy theorem
☐ : User-defined

Program Correctness

Program Logic

Custom Program Logic

Iris Base Logic

Operational Semantics

Coq

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.
A logic with features designed for defining program logics:

\[ P ::= \text{True} \mid \text{False} \mid P \lor P \mid P \land P \mid P \to P \mid \forall x. P \mid \exists x. P \mid \] (higher-order logic)

\[ P \times P \mid \] (separation logic)

\[ \exists \alpha \mid \Rightarrow P \mid \] (user-defined resources)

\[ \triangleright P \mid \mu r. P \mid \] (step indexing)

\[ \neg P \] (invariants)

Base logic inference rules:
Program Logic

A Hoare-style logic:

Examples: \( \{ n \geq 0 \} \) fact \( n \) \( \{ x. \, x = n! \} \) \( \{ \text{True} \} \) letrec \( f \) \( x = f \, x \) inf \( 4 \) \( \{ x. \, \text{False} \} \)
Program Logic

A Hoare-style logic:

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

Examples:

$$\{n \geq 0\} \ fact \ n \ \{x. \ x = n!\} \quad \{True\} \ \text{letrec} \ f \ x = f \ x \ \text{in} \ 4 \ \{x. \ False\}$$

Theorem (Adequacy)

If we prove

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

in Iris for a suitable $\phi$, then $\text{Correct}_\phi(e)$
Program Logic

A Hoare-style logic:

\[ \{P\} e \{x. \ \phi\} \]

precondition ⇋ binder for return value ⇋ postcondition

Examples:

\{n \geq 0\} fact n \{x. \ x = n!\}  \quad \{\text{True}\} \ \text{letrec} \ f \ x = f \ x \ \text{in} \ f \ 4 \ \{x. \ \text{False}\}

Theorem (Adequacy)

If we prove

\[ \vdash \{\text{True}\} e \{x. \ \phi(x)\} \]

in Iris for a suitable \(\phi\), then \(\text{Correct}_{\phi}(e)\)

Proof rules for reasoning about programs:
Expressivity: Higher-Order Logic

Specifying abstract data types:\(^1\)

\[ \exists \text{isStack} : \text{Val} \rightarrow \text{list(Val} \rightarrow \text{Prop}) \rightarrow \text{Prop}. \]
\[ \{\text{True}\} \text{ mk_stack}(\text{)} \{s.\text{isStack}(s, [])\} \land \]
\[ \forall s.\forall \Phi.\forall \Phi s.\{\text{isStack}(s, \Phi s) \ast \Phi(x)\} \text{ push}(x, s) \{v.\; v = () \land \text{isStack}(s, \Phi :: \Phi s)\} \land \]
\[ \forall s.\forall \Phi.\forall \Phi s.\{\text{isStack}(s, \Phi :: \Phi s)\} \text{ pop}(s) \{v.\Phi(v) \ast \text{isStack}(s, \Phi s)\} \]

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:

\[ P \ast Q \]

\( P \ast Q \) holds if \( P \) and \( Q \) hold for disjoint resources

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

\[ \ell \mapsto v \ast \ell' \mapsto v' \land \ell \neq \ell' \]

**Hoare-alloc**

\{True\} \text{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:

\[
\text{HOARE-FRAME} \quad\begin{array}{c}
\{P\} e \{x. Q\} \\
\{P \ast R\} e \{x. Q \ast R\}
\end{array}
\]

**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)

\[ (a) \gamma * (b) \gamma \rightarrow (a - b) \gamma \]

logical equivalence user-defined operation

update modality

\[ \models P \text{ holds if } P \text{ holds after updating resources} \]

Idea: for verifying stateful programs we need a stateful logic
Iris invariants are *impredicative*:

*P* can be any proposition; it may also including invariants

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

\[ LR_\tau(e) \Rightarrow Safe(e) \]

However, we cannot take \( LR_\tau(e) \) to be \( Safe(e) \):

\[ Safe(e_1) \land Safe(e_2) \not\Rightarrow Safe(e_1 - e_2) \]

Counter example: Safe(\texttt{true}) and Safe(3) but \( \neg Safe(\texttt{true} - 3) \)
Expressivity: Step-Indexing and Invariants (Logical Relations)

We should take $LR_\tau$ to be:

$$LR_\tau(e) \triangleq \text{Correct}_{\llbracket \tau \rrbracket}(e)$$

where $\llbracket \tau \rrbracket(v)$ means that $v$ is a value of type $\tau$.

Ideally, we should define this by induction on types:

$$\llbracket \text{int} \rrbracket(v) \triangleq v \in \mathbb{Z}$$

$$\llbracket (\tau_1 \times \tau_2) \rrbracket(v) \triangleq \exists v_1, v_2. v = (v_1, v_2) \land \llbracket \tau_1 \rrbracket(v_1) \land \llbracket \tau_2 \rrbracket(v_2)$$

$$\vdots$$

$$\llbracket \mu X. \tau \rrbracket(v) \triangleq \exists w. v = \text{fold}(w) \land \llbracket \tau \rrbracket_{X \Rightarrow \llbracket \mu X. \tau \rrbracket}(w)$$

$$\llbracket \text{ref}(\tau) \rrbracket(v) \triangleq \exists \ell. v = \ell \land \ell \text{ always stores a value of } \tau$$

how do we express this?
Expressivity: Step-Indexing and Invariants (Logical Relations)

We use Iris and define

\[ \text{LR}_\tau(e) \triangleq \{ \text{True} \} \; e \{ x. \; \llbracket \tau \rrbracket (v) \} \]

We define \( \llbracket \tau \rrbracket (v) \) inductively as follows:

\[
\begin{align*}
\llbracket \text{int} \rrbracket (v) & \triangleq v \in \mathbb{Z} \\
\llbracket (\tau_1 \times \tau_2) \rrbracket (v) & \triangleq \exists v_1, v_2. v = (v_1, v_2) \land \llbracket \tau_1 \rrbracket (v_1) \land \llbracket \tau_2 \rrbracket (v_2) \\
\\vdots
\end{align*}
\]

Iris’s guarded recursion

\[
\begin{align*}
\llbracket \mu X. \tau \rrbracket & \triangleq \mu r. \lambda v. \exists w. v = \text{fold} (w) \land \triangleright \llbracket \tau \rrbracket_{X \mapsto r} (w) \\
\llbracket \text{ref} (\tau) \rrbracket (v) & \triangleq \exists \ell. v = \ell \land \exists w. \ell \mapsto w \ast \llbracket \tau \rrbracket (w)
\end{align*}
\]

may include invariants
Expressivity: Step-Indexing and Invariants (Logical Relations)

\[ [\Xi \vdash a]_\Delta(v, v') \triangleq \Delta(a)(v, v') \]
\[ [\Xi \vdash N]_\Delta(v, v') \triangleq v = v' \in \mathbb{N} \]
\[ [\Xi \vdash t \times t]_\Delta(v, v') \triangleq \exists w, v_1, v_2 : v = (v_0, v_2) \land v' = (v_0, v_2)' \land \]  
\[ [\Xi \vdash t]_\Delta(v, v') \land [\Xi \vdash t]_\Delta(v_0, v_0') \]
\[ [\Xi \vdash t_1 \rightarrow \tau]_\Delta(v, v') \triangleq \forall w, w'. \delta([\Xi \vdash t_1]_\Delta(w, w') \rightarrow [\Xi \vdash t]_\Delta(v, v' w')) \]
\[ [\Xi \vdash \mu z. t]_\Delta(v, v') \triangleq p f. \exists w, w'. v = f o l d w \land v' = f o l d w' \land \]  
\[ [\Xi, \Xi \vdash \mu z. t]_\Delta(\mu z. t)(w, w') \]
\[ [\Xi \vdash r e f(t)]_\Delta(v, v') \triangleq \exists v : v = \ell' \land v' = \ell' \land \]  
\[ \exists w, w', \ell \rightarrow \ell' \rightarrow \ell' \rightarrow [\Xi \vdash t]_\Delta(w, w') \]
\[ [\Xi \vdash r]_\Delta(e, e') \triangleq \exists \psi, k. \{ \text{spec inv}(\psi) + j \mapsto e' \} \]
\[ \varepsilon \]
\[ \{ v, \exists v'. j \mapsto v' \rightarrow [\Xi \vdash t]_\Delta(v, v') \} \]

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.
Consider the following concurrent program where threads share memory:

```plaintext
let c = ref 0 in
(atomic fetch and add operation)
f aa c 1 || f aa c 2 ;
! c
```
Example: Shared Memory Concurrency

Consider the following concurrent program where threads share memory:

{True}

let c = ref 0 in
(faa c 1 || faa c 2);
! c

{x. x ≥ 0}
Example: Shared Memory Concurrency

Consider the following concurrent program where threads share memory:

\{True\}

\begin{verbatim}
let c = ref 0 in
{c \mapsto 0}
\begin{array}{l}
\{\exists n. \; n \geq 0 \; \land \; c \mapsto n\} \\
\{\exists n. \; n \geq 0 \; \land \; c \mapsto n\} \\
\{\exists n. \; n \geq 0 \; \land \; c \mapsto n\} \\
\{x. \; \exists n. \; n \geq 0 \; \land \; c \mapsto n\} \\
\{\exists n. \; n \geq 0 \; \land \; c \mapsto n\} \\
!c
\end{array}
\begin{array}{l}
\{c \mapsto 0\} \\
\{c \mapsto 0\} \\
\{c \mapsto 0\} \\
\{x. \; c \mapsto 0\} \\
\{x. \; c \mapsto 0\} \\
\{x. \; c \mapsto 0\}
\end{array}
\end{verbatim}
Example: Shared Memory Concurrency (Stronger Postcondition)

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

```
{True}
  let c = ref 0 in
  (faa c 1 || faa c 2);
  ! 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?

\[\exists n. n \geq 0 \times c \mapsto n\quad c \mapsto 3\]

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

{True}

```haskell
let c = ref 0 in
(faa c 1 || faa c 2);
! c

{x. x = 3}
```
Example: Shared Memory Concurrency (Stronger Postcondition)

We use user-defined resources to define the following:

- \( \text{Sum}^Y \): Sum of contributions
- \( \text{Left}^Y \): Contributions of the left thread
- \( \text{Right}^Y \): Contributions of the right thread

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

\[
\begin{align*}
\text{Contr-alloc} & \quad \vdash \Rightarrow \exists y. \text{Sum}^Y(0) \ast \text{Left}^Y(0) \ast \text{Right}^Y(0) \\
\text{Contr-sum} & \quad \vdash \text{Sum}^Y(n) \ast \text{Left}^Y(m) \ast \text{Right}^Y(k) \vdash n = m + k \\
\text{Contr-incr-left} & \quad \vdash \Rightarrow \text{Sum}^Y(n) \ast \text{Left}^Y(m) \vdash \Rightarrow \text{Sum}^Y(n + k) \ast \text{Left}^Y(m + k) \\
\text{Contr-incr-right} & \quad \vdash \Rightarrow \text{Sum}^Y(n) \ast \text{Right}^Y(m) \vdash \Rightarrow \text{Sum}^Y(n + k) \ast \text{Right}^Y(m + k)
\end{align*}
\]
Can we also prove the following stronger specs for our code?

\{ \text{True} \}

\begin{verbatim}
let c = ref 0 in 
(faa c 1 || faa c 2); 
! c
\end{verbatim}

\{ x. x = 3 \}
Can we also prove the following stronger specs for our code?

\{ \text{True} \}

\begin{align*}
\text{let } & c = \text{ref } 0 \text{ in } \\
\{ & c \mapsto 0 \}
\end{align*}

\begin{align*}
\{ & c \mapsto 0 \ast \text{Sum}^y(0) \ast \text{Left}^y(0) \ast \text{Right}^y(0) \}
\end{align*}

\begin{align*}
\left\{ & \exists n. \; c \mapsto n \ast \text{Sum}^y(n) \ast \text{Left}^y(0) \ast \text{Right}^y(0) \right\} \\
\left( & \left\{ \exists n. \; c \mapsto n \ast \text{Sum}^y(n) \ast \text{Left}^y(0) \right\} \parallel \left\{ \exists n. \; c \mapsto n \ast \text{Sum}^y(n) \ast \text{Right}^y(0) \right\} \right) \\
& \quad \text{faa } c \ 1 \\
& \quad \{ x. \; \text{Left}^y(1) \} \\
& \quad \text{faa } c \ 2 \\
& \quad \{ x. \; \text{Right}^y(2) \} \\
\left\{ & \exists n. \; c \mapsto n \ast \text{Sum}^y(n) \ast \text{Left}^y(1) \ast \text{Right}^y(2) \right\} \\
\end{align*}

\begin{align*}
! & c \\
\{ & x. \; x = 3 \}
\end{align*}
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
<table>
<thead>
<tr>
<th>Title</th>
<th>Venue</th>
<th>Date</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Toward Complete Stack Safety for Capability Machines</strong></td>
<td>@PriSC</td>
<td>17th</td>
<td>20:12</td>
</tr>
<tr>
<td>Aïna Linn Georges, Armaël Guéneau, Alix Trieu, Lars Birkedal</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Contextual Refinement of the Micheal-Scott Queue (Proof Pearl)</strong></td>
<td>@CPP</td>
<td>18th</td>
<td>17:00</td>
</tr>
<tr>
<td>Amin Timany, Lars Birkedal</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Reasoning About Monotonicity in Separation Logic</strong></td>
<td>@CPP</td>
<td>18th</td>
<td>17:15</td>
</tr>
<tr>
<td>Amin Timany, Lars Birkedal</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>[T4] Iris – A Modular Foundation for Higher-Order Concurrent Separation Logic</strong></td>
<td>@Tutorial</td>
<td>18th</td>
<td>18:00</td>
</tr>
<tr>
<td>Tej Chajed, Ralf Jung, Joseph Tassarotti</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Machine-Checked Semantic Session Typing</strong></td>
<td>@CPP</td>
<td>18th</td>
<td>18:30</td>
</tr>
<tr>
<td>Jonas Kastberg Hinrichsen, Daniël Louwrink, Robbert Krebbers, Jesper Bengtson</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Fully Abstract from Static to Gradual</strong></td>
<td>@POPL</td>
<td>20th</td>
<td>16:00</td>
</tr>
<tr>
<td>Koen Jacobs, Amin Timany, Dominique Devriese</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Efficient and Provable Local Capability Revocation using Uninitialized Capabilities</strong></td>
<td>@POPL</td>
<td>21st</td>
<td>16:00</td>
</tr>
<tr>
<td>Aïna Linn Georges, Armaël Guéneau, Thomas Van Strydonck, Amin Timany, Alix Trieu, Sander Huyghebaert, Dominique Devriese, Lars Birkedal</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Mechanized Logical Relations for Termination-Insensitive Noninterference</strong></td>
<td>@POPL</td>
<td>21st</td>
<td>16:10</td>
</tr>
<tr>
<td>Simon Oddershede Gregersen, Johan Bay, Amin Timany, Lars Birkedal</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>A Separation Logic for Effect Handlers</strong></td>
<td>@POPL</td>
<td>22nd</td>
<td>16:15</td>
</tr>
<tr>
<td>Paulo Emílio de Vilhena, François Pottier</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Distributed Causal Memory: Modular Specification and Verification in Higher-Order Distributed Separation Logic</strong></td>
<td>@POPL</td>
<td>22nd</td>
<td>16:15</td>
</tr>
<tr>
<td>Léon Gondelman, Simon Oddershede Gregersen, Abel Nieto, Amin Timany, Lars Birkedal</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>