A Formal and Foundational Approach to Program Verification for Safety and Security

Amin Timany

Aarhus University,
Aarhus, Denmark

Sep 20–22, 2022,
Summer School on Security Testing and Verification,
Leuven, Belgium

These slides: https://cs.au.dk/~timany/talks/leuvenss22

Introduction

It is important to make sure that critical software systems are safe and secure

Our approach: formal proof of safety and security properties of programs and PL's

We use mathematical tools:

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

This is very similar to what other engineers do

- **They build a mathematical model of the building/structure they are planning**
- **Analyze the model to make sure it is resilient against, *e.g.*, earthquakes**

Introduction

Program logics are important tools

- **Based on mathematical logic**
- **Provide a formal framework for stating and proving properties of programs**
- **In this course: an overview of the Iris program logic and its applications**

Programs' semantics

In order to determine whether a program is correct/safe/secure we need to understand its meaning (semantics).[1]

We use small-step operational semantics:

- A mathematical relation $\longrightarrow$ describing individual steps of computation.
- We write $\longrightarrow^*$ for zero or more steps of computation

  Formally, this is the reflexive transitive closure of $\longrightarrow$

Example:

$2 + 3 \longrightarrow 5$

$(2 + 3 + 7) * 2 \longrightarrow^* 24$; in details: $(2 + 3 + 7) * 2 \longrightarrow (5 + 7) * 2 \longrightarrow 12 * 2 \longrightarrow 24$

[1] What we present here is slightly simplified. Semantics needs to also take into account the state of the machine, *e.g.*, contents of memory.

Programs' semantics

We distinguish a class of expressions called values:

- These are *values* we expect as the end result of computations
- Examples: numerals (2, 3, *etc.*), booleans, memory locations (references/pointers), functions, *etc.*
- Non-examples: 2 + 3, "a" - 3, 4 "a", $\ell[10]$, $!\,\ell$ *etc.*

In this formalism we characterize errors (program crashing) as stuck programs:

- These are programs that are neither values nor can they take any step of computation
- Examples: "a" - 3 (treating a string as a number), 4 "a" (treating a number as a function), *etc.*
- How about $\ell[10]$ and $!\,\ell$? Are these programs stuck?

Programs' semantics

We distinguish a class of expressions called values:

- These are *values* we expect as the end result of computations
- Examples: numerals (2, 3, *etc.*), booleans, memory locations (references/pointers), functions, *etc.*
- Non-examples: 2 + 3, "a" - 3, 4 "a", $\ell[10]$, $!\,\ell$ *etc.*

In this formalism we characterize errors (program crashing) as stuck programs:

- These are programs that are neither values nor can they take any step of computation
- Examples: "a" - 3 (treating a string as a number), 4 "a" (treating a number as a function), *etc.*
- How about $\ell[10]$ and $!\,\ell$? Are these programs stuck?

It depends on the contents of the memory.
These programs could result in memory violations.

Some Interesting Properties

Safety: program does not crash

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

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

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

Some Interesting Properties

Safety: program does not crash

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

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

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

Functional Correctness: safe, and upon termination postcondition holds

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

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

Some Interesting Properties

Safety: program does not crash

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

— 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

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

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

Type safety: well-typed programs are safe

Is safety interesting?

Does safety, *i.e.*, programs not crashing, have security implications?

Yes, many security vulnerabilities arise as safety (memory) violations, *e.g.*, the infamous Heardbleed bug.

An aside: there are other interesting properties that our methodology supports but are not covered in this course, *e.g.*, non-interference.

Example of Security Vulnerability in Implementation: Heartbleed

A bug in OpenSSL's implementation of the heartbeat feature:

- One side sends a heartbeat request message m together with a number l
- The other side sends the first l characters of m back to signal that it is alive

Example of Security Vulnerability in Implementation: Heartbleed

A bug in OpenSSL's implementation of the heartbeat feature:

- One side sends a heartbeat request message m together with a number l
- The other side sends the first l characters of m back to signal that it is alive

A simplified version of implementation:

```
void answer_heartbeat(SSL *req, unsigned int l){
  send_reply(l, req->data);
}
```

Example of Security Vulnerability in Implementation: Heartbleed

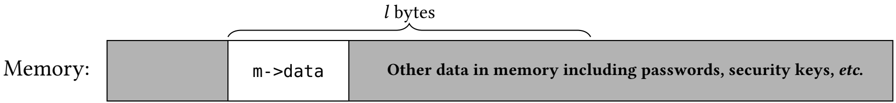
A bug in OpenSSL's implementation of the heartbeat feature:

- One side sends a heartbeat request message m together with a number l
- The other side sends the first l characters of m back to signal that it is alive

A simplified version of implementation:

```
void answer_heartbeat(SSL *req, unsigned int l){
  send_reply(l, req->data);
}
```

What happens if $l > length(m)$?



Example of Security Vulnerability in Implementation: Heartbleed

A bug in OpenSSL's implementation of the heartbeat feature:

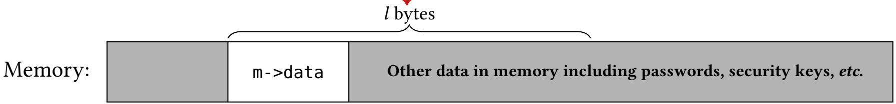
- One side sends a heartbeat request message m together with a number l
- The other side sends the first l characters of m back to signal that it is alive

A simplified version of implementation:

```
void answer_heartbeat(SSL *req, unsigned int l){
  send_reply(l, req->data);
}
```

What happens if $l > length(m)$?

This is a *memory violation* and **would have been caught** had the program been verified.



Example of Security Vulnerability in Implementation: Heartbleed

A bug in OpenSSL's implementation of the heartbeat feature:

- One side sends a heartbeat request message m together with a number l
- The other side sends the first l characters of m back to signal that it is alive

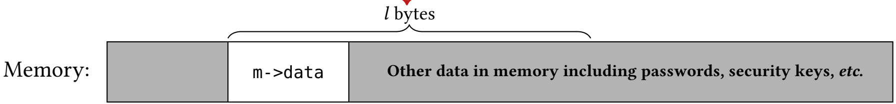
A simplified version of implementation:

```
void answer_heartbeat(SSL *req, unsigned int l){
  if(l > req->length){return;}
  send_reply(l, req->data);
}
```

The fix

What happens if $l > length(m)$?

This is a *memory violation* and **would have been caught** had the program been verified.



Challenges

We defined safety as a desirable property to prove about programs.

Question: How do we reason about safety of large programs based on a detailed operational semantics?

– There are many details, especially when we consider concurrent and distributed systems

A foundational approach, *i.e.*, based on first principles, in a proof assistant (Coq)

The Proof Assistant Coq

A proof assistant based on the Calculus of Inductive Constructions

- Coq is itself a programming language:
  - Curry-Howard correspondence (types are theorems, programs are proofs)
  - It has an interesting meta-theory called *type theory*
- Proofs written and checked against foundational mathematical principles:
  - Coq only understands functions and the concept of induction

An example:

- Commutativity of addition for natural numbers
- Proof automation can help but still this demonstrates the level of formality

```
Theorem add_com n m : n + m = m + n.
Proof.
  revert m.
  induction n as [|n IHn].
  - intros m.
    simpl.
    induction m as [|m IHm].
    + simpl. trivial.
    + simpl. rewrite <- IHm. reflexivity.
  - intros m.
    induction m as [|m IHm].
    + simpl.
      rewrite IHn. simpl. reflexivity.
    + simpl.
      rewrite IHn.
      simpl.
      rewrite <- IHm.
      simpl.
      rewrite IHn.
      reflexivity.
Qed.
```

Proof assistants are the highest standard of rigor for mathematical proofs

The Proof Assistant Coq

We use Coq to reason about state-of-the-art programs and programming languages:

- We define the precise mathematical model (operational semantics) of program execution
- The level of details in these models necessitates the use of proof assistants and program logics
- We define program logics (*the Iris framework*) for these programs
- Use these to prove correctness of programs

Challenges

Question: How do we reason about safety of a large programs based on a detailed operational semantics at this level of detail in Coq?

- Coq solves the problem of mathematical rigor
- Still, proofs in Coq are not easier than those on paper; they are in fact more detailed and longer ...
- How do we manage the complexity of proofs?

Abstraction and Modularity

Abstraction and Modularity

Abstraction and modularity are important related concepts whereby we mean:

- Abstract reasoning: hiding details not relevant to the core of the problem at hand, *e.g.*
  - individual steps of computation
  - scheduler (in case of concurrency)
  - the contents of the (entire) memory
  - networking layer (in case of distributed systems)
- Modular reasoning: composing of proofs of separate modules to prove correctness of composed modules, *e.g.*
  - modular specs for libraries, *e.g.*, abstract specs for ADT's like stacks
  - reasoning about different threads in isolation
  - only considering a module's memory footprint, *i.e.*, parts the module might touch
  - reasoning about different nodes in the network in isolation

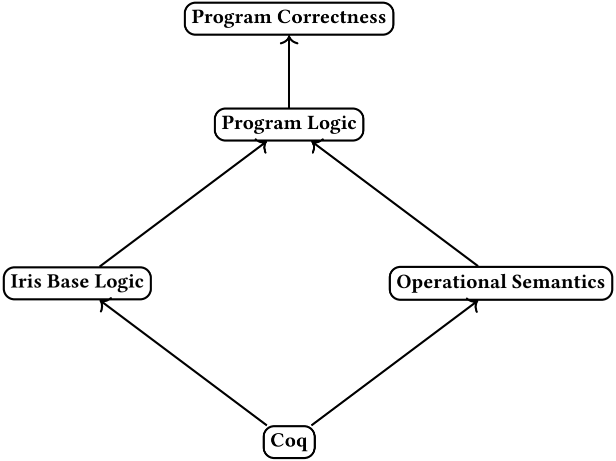
Modularity is also important for robust safety as we will see.

Abstraction and modularity are things that a program logic gives us.

What is Iris?

A Framework for Higher-order Concurrent Separation Logics

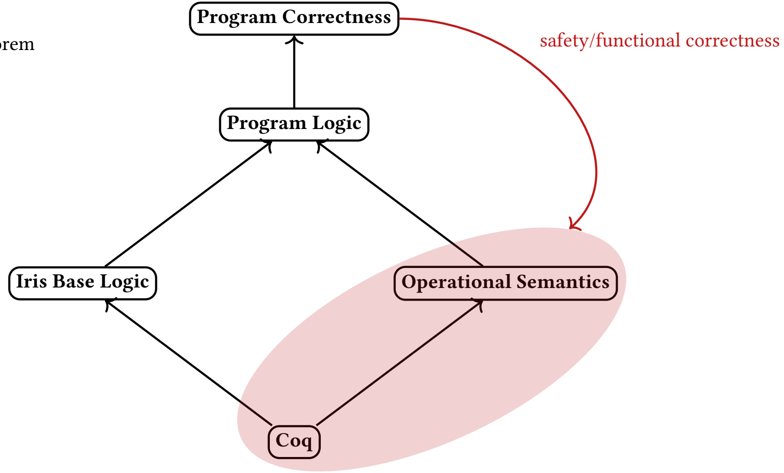
→ : Built on top of



What is Iris?

A Framework for Higher-order Concurrent Separation Logics

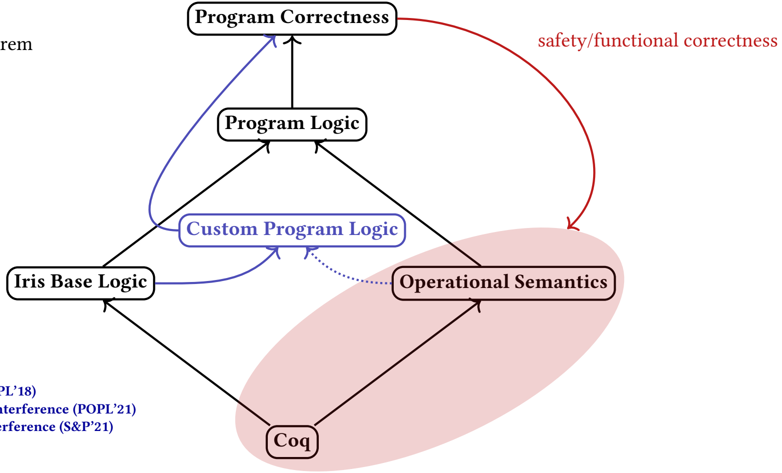
→ : Built on top of
→ : Iris's adequacy theorem



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 has been used in many projects:

- Reasoning about session types
- Reasoning about capability machines (hardware language)
- Reasoning about non-interference (a security property)
- Reasoning about distributed systems
- Proving properties of gradual typing systems
- Reasoning about algebraic effect handlers
- 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:

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

Base logic inference rules:

$$\textsc{Löb-ind}\ \frac{\triangleright P \vdash P}{\vdash P} \qquad \triangleright\textsc{-intro}\ P \vdash \triangleright P \qquad \triangleright\textsc{-mono}\ \frac{P \vdash Q}{\triangleright P \vdash \triangleright Q} \qquad *\textsc{-true}\ P * \mathsf{True} \dashv\vdash P \qquad *\textsc{-elim-L}\ P * Q \vdash P \qquad *\textsc{-elim-R}\ P * Q \vdash Q$$

$$*\textsc{-mono}\ \frac{P_1 \vdash P_2 \qquad Q_1 \vdash Q_2}{P_1 * Q_1 \vdash P_2 * Q_2} \qquad *\textsc{-comm}\ P * Q \vdash Q * P \qquad *\textsc{-assoc}\ (P * Q) * R \vdash P * (Q * R) \qquad \land\textsc{-intro}\ \frac{P \vdash Q \qquad P \vdash R}{P \vdash Q \land R}$$

$$\vdash\textsc{-refl}\ P \vdash P \qquad \vdash\textsc{-trans}\ \frac{P \vdash Q \qquad Q \vdash R}{P \vdash R} \qquad \vdash\textsc{-true}\ P \vdash \mathsf{True} \qquad \vdash\textsc{-falso}\ \mathsf{False} \vdash P \qquad \land\textsc{-true}\ P \land \mathsf{True} \dashv\vdash P$$

$$\land\textsc{-elim-L}\ P \land Q \vdash P \qquad \land\textsc{-elim-R}\ P \land Q \vdash Q \qquad \land\textsc{-mono}\ \frac{P_1 \vdash P_2 \qquad Q_1 \vdash Q_2}{P_1 \land Q_1 \vdash P_2 \land Q_2} \qquad \land\textsc{-comm}\ P \land Q \vdash Q \land P \qquad \land\textsc{-assoc}\ (P \land Q) \land R \vdash P \land (Q \land R)$$

$$\lor\textsc{-intro-L}\ \frac{P \vdash Q}{P \vdash Q \lor R} \qquad \lor\textsc{-intro-R}\ \frac{P \vdash R}{P \vdash Q \land R} \qquad \lor\textsc{-false}\ P \lor \mathsf{False} \dashv\vdash P \qquad \lor\textsc{-elim}\ \frac{P \vdash R \qquad Q \vdash R}{P \lor Q \vdash R}$$

$$\lor\textsc{-mono}\ \frac{P_1 \vdash P_2 \qquad Q_1 \vdash Q_2}{P_1 \lor Q_1 \vdash P_2 \lor Q_2} \qquad \lor\textsc{-comm}\ P \lor Q \vdash Q \lor P \qquad \lor\textsc{-assoc}\ (P \lor Q) \lor R \vdash P \lor (Q \lor R)$$

Program Logic

A Hoare-style logic:

program

binder for return value

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

precondition

postcondition

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

Program Logic

A Hoare-style logic:

program

binder for return value

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

precondition

postcondition

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

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:

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

program — binder for return value — precondition — postcondition

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

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
$$\frac{\{P * \boxed{R}\}\ e\ \{x.\ Q\}}{\{P * R\}\ e\ \{x.\ Q\}}$$

Specifying abstract data types:[2]

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

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

[2] Taken verbatim from Iris lecture notes.

Expressivity: Separation Logic

Separating conjunction:

separating conjunction

$$P * Q$$

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

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

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

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

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

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

Expressivity: Separation Logic

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

Hence the *frame* and *par* rules:

Hoare-Frame

$$\frac{\{P\}\; e\; \{x.\; Q\}}{\{P * R\}\; e\; \{x.\; Q * R\}}$$

Hoare-par

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

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

$\Rrightarrow P$ **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*:

$\boxed{P}$

P can be any proposition;
it may also include 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

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

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:

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

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

Ideally, we should define this by induction on types:

$$\begin{aligned}
\llbracket int \rrbracket(v) &\triangleq v \in \mathbb{Z} \\
\llbracket (\tau_1 \times \tau_2) \rrbracket(v) &\triangleq \exists v1, v_2. v = (v_1, v_2) \wedge \llbracket \tau_1 \rrbracket(v_1) \wedge \llbracket \tau_2 \rrbracket(v_2) \\
\llbracket \tau_1 \to \tau_2 \rrbracket(f) &\triangleq \forall v. \{\llbracket \tau_1 \rrbracket(w)\}\, f\ v\, \{x.\ \llbracket \tau_2 \rrbracket(x)\} \\
&\vdots \\
\llbracket \mu X. \tau \rrbracket(v) &\triangleq \exists w.\ v = \mathtt{fold}\,(w) \wedge \llbracket \tau \rrbracket_{X \mapsto \llbracket \mu X. \tau \rrbracket}(w) \\
\llbracket \mathtt{ref}(\tau) \rrbracket(v) &\triangleq \exists \ell.\ v = \ell \wedge \underline{\ell \text{ always stores a value of } \tau}
\end{aligned}$$

circular definition

how do we express this?

We use Iris and define

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

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

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

Iris's guarded recursion

may include invariants

Expressivity: Step-Indexing and Invariants (Logical Relations)

$$\llbracket \Xi \vdash \alpha \rrbracket_\Delta(v, v') \triangleq \Delta(\alpha)(v, v')$$

$$\llbracket \Xi \vdash \mathbb{N} \rrbracket_\Delta(v, v') \triangleq v = v' \in \mathbb{N}$$

$$\llbracket \Xi \vdash \tau_1 \times \tau_2 \rrbracket_\Delta(v, v') \triangleq \exists v_1, v_2, v_1', v_2'.\, v = (v_1, v_2) \wedge v' = (v_1', v_2') \wedge \llbracket \Xi \vdash \tau_1 \rrbracket_\Delta(v_1, v_1') \wedge \llbracket \Xi \vdash \tau_2 \rrbracket_\Delta(v_2, v_2')$$

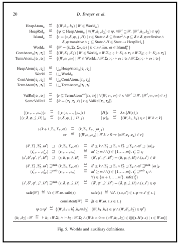
$$\llbracket \Xi \vdash \tau_1 \to \tau_2 \rrbracket_\Delta(v, v') \triangleq \forall w, w'.\, \square(\llbracket \Xi \vdash \tau_1 \rrbracket_\Delta(w, w') \twoheadrightarrow \mathcal{E}\llbracket \Xi \vdash \tau_2 \rrbracket_\Delta(v\ w, v'\ w'))$$

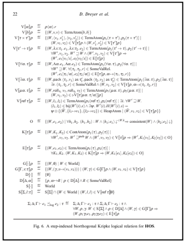
$$\llbracket \Xi \vdash \forall \alpha.\, \tau \rrbracket_\Delta(v, v') \triangleq \forall f.\, \square(\llbracket \alpha, \Xi \vdash \tau \rrbracket_{\Delta[\alpha \mapsto f]}(v\ \_, v'\ \_))$$

$$\llbracket \Xi \vdash \mu\alpha.\, \tau \rrbracket_\Delta(v, v') \triangleq \mu f.\, \exists w, w'.\, v = \texttt{fold}\ w \wedge v' = \texttt{fold}\ w' \wedge \triangleright \llbracket \alpha, \Xi \vdash \tau \rrbracket_{\Delta[\alpha \mapsto f]}(w, w')$$

$$\llbracket \Xi \vdash \texttt{ref}(\tau) \rrbracket_\Delta(v, v') \triangleq \exists \ell.\, v = \ell \wedge v' = \ell' \wedge \boxed{\exists w, w'.\, \ell \mapsto w * \ell' \mapsto w' * \llbracket \Xi \vdash \tau \rrbracket_\Delta(w, w')}^{\mathcal{N}.\ell}$$

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


20 D. Dreyer et al.
Fig. 5. Worlds and auxiliary definitions.


22 D. Dreyer et al.
Fig. 6. A step-indexed biorthogonal Kripke logical relation for HOS.

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:

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

atomic *fetch and add* operation

Example: Shared Memory Concurrency

Consider the following concurrent program where threads share memory:

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

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

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

Example: Shared Memory Concurrency

Consider the following concurrent program where threads share memory:

$$
\begin{array}{l}
\{\mathsf{True}\} \\
\quad \texttt{let}\ c = \texttt{ref}(0)\ \texttt{in} \\
\quad \{c \mapsto 0\} \\
\quad \{\boxed{\exists n.\ n \geq 0 * c \mapsto n}\} \\
\quad \left(\begin{array}{l}
\{\boxed{\exists n.\ n \geq 0 * c \mapsto n}\} \\
\{\exists n.\ n \geq 0 * c \mapsto n\} \\
\quad \texttt{faa}\ c\ 1 \\
\{x.\ \exists n.\ n \geq 0 * c \mapsto n\}
\end{array}
\middle\|
\begin{array}{l}
\{\boxed{\exists n.\ n \geq 0 * c \mapsto n}\} \\
\{\exists n.\ n \geq 0 * c \mapsto n\} \\
\quad \texttt{faa}\ c\ 2 \\
\{x.\ \exists n.\ n \geq 0 * c \mapsto n\}
\end{array}\right); \\
\quad \{\boxed{\exists n.\ n \geq 0 * c \mapsto n}\} \\
\quad !\,c \\
\{x.\ x \geq 0\}
\end{array}
$$

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?

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)

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

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

$\{\mathsf{True}\}$

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

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

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?

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

```
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?

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

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

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

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

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

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

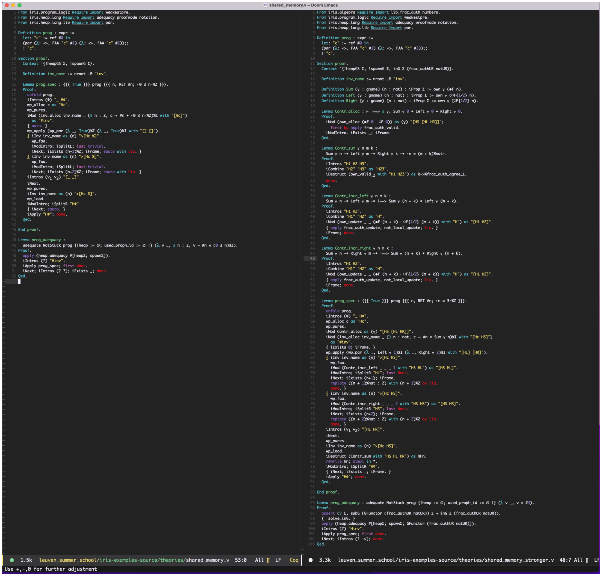
$$\{\boxed{\exists n.\ c \mapsto n * \mathsf{Sum}^{\gamma}(n)} * \mathsf{Left}^{\gamma}(1) * \mathsf{Right}^{\gamma}(2)\}$$

$$!\,c$$

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

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 using IPM



Robust Safety

Our modular reasoning principles imply that we can combine programs that are proven, *e.g.* the Hoare-par rule.

Question: What can we say about combining a proven correct program with an arbitrary, adversarial program?

Important insight:

- We can express limitations of arbitrary programs in terms Iris propositions and Hoare triples.
- Modular reasoning: we can combine these Hoare triples with those of proven correct programs.
- We obtain proofs about the result of linking a proven correct program with an arbitrary, adversarial program.

Robust Safety

Notice: We consider arbitrary programs which may crash.

Hence, we use a weaker, non-progressive variant of Hoare triples which allow the program to get stuck:

$$\{P\}_{\lightning}\; e\, \{x.\ Q\}$$

Just like ordinary Hoare triples the non-progressive version also enforces invariants and does not allow assertion (we will see) failures.

$$\{P\}\, e\, \{x.\ Q\} \vdash \{P\}_{\lightning}\; e\, \{x.\ Q\}$$

All the modular reasoning principles of ordinary Hoare triples, *e.g.*, Hoare-frame, Hoare-par, *etc.*, also hold for the non-progressive variant.

Robust Safety

Similar to $\mathrm{Correct}_\phi$ we define $\mathrm{NonProg}_\phi$ which

- does not guarantee progress (programs may get stuck)
- requires no assertion failures
- if the program terminates to a value v, $\phi(v)$ must hold

Theorem (Non-progressive Adequacy)

If we prove

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

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

Robust Safety, an Example

The following *even_counter* module returns two functions, one to read the value and one to increment it by two.

$$\begin{aligned}even\_counter \triangleq\ & \texttt{let}\ c = \texttt{ref}(0)\ \texttt{in} \\ & \texttt{let}\ incr\ \_ = \texttt{faa}\ c\ 2\ \texttt{in} \\ & \texttt{let}\ read\ \_ = \texttt{let}\ x = !\, c\ \texttt{in}\ \texttt{assert}(x \,\%\, 2 = 0);\ x\ \texttt{in} \\ & (incr, read)\end{aligned}$$

Question: can we prove $\mathrm{NonProg}_{isEven}(prog)$?

$$prog = \texttt{let}\ (incr, read) = even\_counter\ \texttt{in}\ adversary;\ \ read\ ()$$

where adversary is a program with no hard-coded locations or assertions.

Robust Safety, an Example

The following *even_counter* module returns two functions, one to read the value and one to increment it by two.

$$
\begin{aligned}
\mathit{even\_counter} \triangleq\ & \mathtt{let}\ c = \mathtt{ref}(0)\ \mathtt{in} \\
& \mathtt{let}\ \mathit{incr}\ \_ = \mathtt{faa}\ c\ 2\ \mathtt{in} \\
& \mathtt{let}\ \mathit{read}\ \_ = \mathtt{let}\ x = !\,c\ \mathtt{in}\ \mathtt{assert}(x \,\%\, 2 = 0);\ x\ \mathtt{in} \\
& (\mathit{incr}, \mathit{read})
\end{aligned}
$$

Question: can we prove $\mathrm{NonProg}_{\mathit{isEven}}(\mathit{prog})$?

$$\mathit{prog} = \mathtt{let}\ (\mathit{incr}, \mathit{read}) = \mathit{even\_counter}\ \mathtt{in}\ \mathit{adversary};\ \ \mathit{read}\ ()$$

where adversary is a program with no hard-coded locations or assertions.

Hint: the language does not support pointer arithmetic; the only way to get a pointer is if the program allocates it or if it is passed to it from another part of the program.

Robust Safety, an Example

Question: is our assumption of no pointer arithmetic reasonable?

Yes, this property holds for our high-level programming language. Hence, we can indeed prove $\mathrm{NonProg}_{isEven}(prog)$ from the previous slide.

Question: But more realistically, programs can be linked to adversary programs written in more low-level languages, *e.g.*, directly in assembly. Surely, those can perform pointer arithmetic.

Yes, however, some modern hardware architectures (still mostly in research labs) feature so-called *capabilities* which essentially restrict pointer arithmetic which can be used to enable the guarantee that we have assumed about pointers.

See our work on studying the assembly language capability machines in Iris for more details.

How Do We Prove Robust Safety?

Question: how do we prove $\text{NonProg}_{isEven}(prog)$?

$$prog = \texttt{let}\ (incr, read) = even\_counter\ \texttt{in}\ adversary;\ \ read\ ()$$

where adversary is a program with no hard-coded locations or assertions.

How Do We Prove Robust Safety?

Question: how do we prove $\mathrm{NonProg}_{isEven}(prog)$?

$$prog = \texttt{let}\ (incr, read) = even\_counter\ \texttt{in}\ adversary;\ \ read\ ()$$

where adversary is a program with no hard-coded locations or assertions.

Modular Reasoning!

How Do We Prove Robust Safety?

We can easily show the following specs for *even_counter*:

$$
\begin{aligned}
&\{\mathsf{True}\}_{\lightning} \\
&\quad \mathit{even\_counter} \\
&\left\{\begin{aligned} &x.\ \exists f, g.\ x = (f, g) \land \\ &\quad (\forall v.\ \{\mathsf{True}\}_{\lightning}\ f\ \ v\,\{y.\ y = ()\}) \land \\ &\quad (\forall v.\ \{\mathsf{True}\}_{\lightning}\ g\ \ v\,\{y.\ \mathit{isEven}(y)\}) \end{aligned}\right\}
\end{aligned}
$$

The proof is very similar to the example before. We simply use an invariant that asserts the location is always even.

If we somehow had a non-progressive Hoare triple for the adversary program, we could just compose it with the spec above; because modularity!

Logical Relations for Establishing Robust Safety

We define a logical relations model for our language:

- We define relations capturing good values and good expressions
- Similar to the logical relations we saw before except relations are not indexed by types
  - Our language has not typed
  - Arbitrary adversarial programs may not be well-typed even if had types
- We show that all adversarial programs (no hard-coded locations or assertions) are good

We define the $good_{val}$ and $good_{exp}$ relations as follows:

$$good_{exp}(e) \triangleq \{\mathsf{True}\}_{\lightning}\ e\,\{x.\ good_{val}(x)\}$$

where $good_{val}(v)$ is inductively as follows:[3]

$$
\begin{aligned}
good_{val}(n) &\triangleq \mathsf{True} && \text{if } n \in \mathbb{Z}\\
good_{val}(b) &\triangleq \mathsf{True} && \text{if } b \in \{\mathtt{true}, \mathtt{false}\}\\
good_{val}(()) &\triangleq \mathsf{True}\\
good_{val}(v_1, v_2) &\triangleq good_{val}(v_1) \wedge good_{val}(v_2)\\
good_{val}(\mathtt{rec}\, f\ x = e) &\triangleq \forall v.\ \{good_{val}(v)\}\ (\mathtt{rec}\, f\ x = e)\ v\,\{x.\ good_{val}(x)\}\\
&\ \ \vdots\\
good_{val}(\ell) &\triangleq \boxed{\exists v.\ \ell \mapsto v * good_{val}(v)}
\end{aligned}
$$

[3]This time by induction on the form of values instead of types.

We define the $good_{val}$ and $good_{exp}$ relations as follows:

$$good_{exp}(e) \triangleq \{\mathsf{True}\}_{\lightning}\ e\, \{x.\ good_{val}(x)\}$$

where $good_{val}(v)$ is inductively as follows:[3]

$$\begin{aligned}
good_{val}(n) &\triangleq \mathsf{True} && \text{if } n \in \mathbb{Z} \\
good_{val}(b) &\triangleq \mathsf{True} && \text{if } b \in \{\mathtt{true}, \mathtt{false}\} \\
good_{val}(()) &\triangleq \mathsf{True} \\
good_{val}(v_1, v_2) &\triangleq good_{val}(v_1) \wedge good_{val}(v_2) \\
good_{val}(\mathtt{rec}\, f\ x = e) &\triangleq \forall v.\ \{good_{val}(v)\}\ (\mathtt{rec}\, f\ x = e)\ v\, \{x.\ good_{val}(x)\} \\
&\vdots \\
good_{val}(\ell) &\triangleq \boxed{\exists v.\ \ell \mapsto v * good_{val}(v)}
\end{aligned}$$

Question: why are these relations not trivial?

[3]This time by induction on the form of values instead of types.

Theorem (Fundamental Theorem of Robust Safety (FTRS))

Let e be any expression with no hard-coded locations or assertions.
Furthermore, let $\overrightarrow{vs}$ be values which are all good.
The following holds:

$$good_{exp}(e[\overrightarrow{vs}/\overrightarrow{xs}])$$

where $\overrightarrow{xs}$ are variables (as many as $\overrightarrow{vs}$).

Proof.

By induction on e. □

Robust Safety, an Example (proof)

$$\{\mathsf{True}\}_{\lightning}\ even\_counter\ \left\{\begin{array}{l} x.\ \exists f, g.\ x = (f, g) \land \\ \quad (\forall v.\ \{\mathsf{True}\}_{\lightning}\ f\ v\ \{y.\ y = ()\}) \land \\ \quad (\forall v.\ \{\mathsf{True}\}_{\lightning}\ g\ v\ \{y.\ isEven(y)\}) \end{array}\right\}$$

$$\begin{array}{l} \{\mathsf{True}\}_{\lightning} \\ \quad \texttt{let}\ (incr, read) = even\_counter\ \texttt{in} \\ \quad adversary; \\ \quad read\ () \\ \{x.\ isEven(x)\} \end{array}$$

Theorem (Fundamental Theorem of Robust Safety (FTRS))

Let e be any expression with no hard-coded locations or assertions. Furthermore, let $\vec{vs}$ be values which are all good. The following holds:

$$good_{exp}(e[\vec{vs}/\vec{xs}])$$

where $\vec{xs}$ are variables (as many as $\vec{vs}$).

Robust Safety, an Example (proof)

$$\{\mathsf{True}\}_{\lightning}$$

$$\left\{\begin{array}{l}(\forall v.\ \{\mathsf{True}\}_{\lightning}\ f\ v\ \{y.\ y = ()\}) \land \\ (\forall v.\ \{\mathsf{True}\}_{\lightning}\ g\ v\ \{y.\ \mathit{isEven}(y)\})\end{array}\right\}_{\lightning}$$

$$\texttt{let}\ (\mathit{incr}, \mathit{read}) = (f, g)\ \texttt{in}$$

$$\mathit{adversary};$$

$$\mathit{read}\ ()$$

$$\{x.\ \mathit{isEven}(x)\}$$

$$\{x.\ \mathit{isEven}(x)\}$$

$$\{\mathsf{True}\}_{\lightning}$$

$$\mathit{even\_counter}$$

$$\left\{\begin{array}{l}x.\ \exists f, g.\ x = (f, g) \land \\ \quad (\forall v.\ \{\mathsf{True}\}_{\lightning}\ f\ v\ \{y.\ y = ()\}) \land \\ \quad (\forall v.\ \{\mathsf{True}\}_{\lightning}\ g\ v\ \{y.\ \mathit{isEven}(y)\})\end{array}\right\}$$

Theorem (Fundamental Theorem of Robust Safety (FTRS))

Let e be any expression with no hard-coded locations or assertions. Furthermore, let $\vec{vs}$ be values which are all good. The following holds:

$$\mathit{good}_{exp}(e[\vec{vs}/\vec{xs}])$$

where $\vec{xs}$ are variables (as many as $\vec{vs}$).

Robust Safety, an Example (proof)

$$\{\mathsf{True}\}_{\lightning}$$

$$\left\{\begin{array}{l}(\forall v.\ \{\mathsf{True}\}_{\lightning}\ f\ v\ \{y.\ y = ()\}) \land \\ (\forall v.\ \{\mathsf{True}\}_{\lightning}\ g\ v\ \{y.\ isEven(y)\})\end{array}\right\}_{\lightning}$$

$$adversary[f, g / incr, read];$$

$$g\ ()$$

$$\{x.\ isEven(x)\}$$

$$\{x.\ isEven(x)\}$$

$$\{\mathsf{True}\}_{\lightning}$$

$$even\_counter$$

$$\left\{\begin{array}{l}x.\ \exists f, g.\ x = (f, g) \land \\ \quad (\forall v.\ \{\mathsf{True}\}_{\lightning}\ f\ v\ \{y.\ y = ()\}) \land \\ \quad (\forall v.\ \{\mathsf{True}\}_{\lightning}\ g\ v\ \{y.\ isEven(y)\})\end{array}\right\}$$

Theorem (Fundamental Theorem of Robust Safety (FTRS))

Let e be any expression with no hard-coded locations or assertions.
Furthermore, let $\vec{vs}$ be values which are all good.
The following holds:

$$good_{exp}(e[\vec{vs}/\vec{xs}])$$

where $\vec{xs}$ are variables (as many as $\vec{vs}$).

Robust Safety, an Example (proof)

$$\{\mathsf{True}\}_{\lightning}$$

$$\left\{\begin{array}{l}(\forall v.\ \{\mathsf{True}\}_{\lightning}\ f\ v\,\{y.\ y = ()\})\ \land \\ (\forall v.\ \{\mathsf{True}\}_{\lightning}\ g\ v\,\{y.\ \mathit{isEven}(y)\})\end{array}\right\}_{\lightning}$$

$$\mathit{adversary}[f, g/\mathit{incr}, \mathit{read}];$$

$$\left\{\begin{array}{l}(\forall v.\ \{\mathsf{True}\}_{\lightning}\ f\ v\,\{y.\ y = ()\})\ \land \\ (\forall v.\ \{\mathsf{True}\}_{\lightning}\ g\ v\,\{y.\ \mathit{isEven}(y)\})\end{array}\right\}$$

$$g\ ()$$

$$\{x.\ \mathit{isEven}(x)\}$$

$$\{x.\ \mathit{isEven}(x)\}$$

$$\{\mathsf{True}\}_{\lightning}$$

$$\mathit{even\_counter}$$

$$\left\{\begin{array}{l}x.\ \exists f, g.\ x = (f, g)\ \land \\ \quad (\forall v.\ \{\mathsf{True}\}_{\lightning}\ f\ v\,\{y.\ y = ()\})\ \land \\ \quad (\forall v.\ \{\mathsf{True}\}_{\lightning}\ g\ v\,\{y.\ \mathit{isEven}(y)\})\end{array}\right\}$$

Theorem (Fundamental Theorem of Robust Safety (FTRS))

Let e be any expression with no hard-coded locations or assertions.
Furthermore, let $\vec{vs}$ be values which are all good.
The following holds:

$$\mathit{good}_{exp}(e[\vec{vs}/\vec{xs}])$$

where $\vec{xs}$ are variables (as many as $\vec{vs}$).

Robust Safety, an Example (proof)

$$\{\mathsf{True}\}_{\lightning}$$
$$\left\{\begin{array}{l}(\forall v.\ \{\mathsf{True}\}_{\lightning}\ f\ v\,\{y.\ y = ()\}) \wedge \\ (\forall v.\ \{\mathsf{True}\}_{\lightning}\ g\ v\,\{y.\ isEven(y)\})\end{array}\right\}_{\lightning}$$
$$adversary[f, g / incr, read];$$
$$\left\{\begin{array}{l}(\forall v.\ \{\mathsf{True}\}_{\lightning}\ f\ v\,\{y.\ y = ()\}) \wedge \\ (\forall v.\ \{\mathsf{True}\}_{\lightning}\ g\ v\,\{y.\ isEven(y)\})\end{array}\right\}$$
$$g\ ()$$
$$\{x.\ isEven(x)\}$$
$$\{x.\ isEven(x)\}$$

$$\{\mathsf{True}\}_{\lightning}$$
$$even\_counter$$
$$\left\{\begin{array}{l}x.\ \exists f, g.\ x = (f, g) \wedge \\ \quad (\forall v.\ \{\mathsf{True}\}_{\lightning}\ f\ v\,\{y.\ y = ()\}) \wedge \\ \quad (\forall v.\ \{\mathsf{True}\}_{\lightning}\ g\ v\,\{y.\ isEven(y)\})\end{array}\right\}$$

Theorem (Fundamental Theorem of Robust Safety (FTRS))

Let e be any expression with no hard-coded locations or assertions. Furthermore, let $\vec{vs}$ be values which are all good. The following holds:

$$good_{exp}(e[\vec{vs}/\vec{xs}])$$

where $\vec{xs}$ are variables (as many as $\vec{vs}$).

Hence, by applying the adequacy theorem for non-progressive Hoare triples, we get

$$NonProg_{isEven}(\texttt{let}\ (incr, read) = even\_counter\ \texttt{in}\ adversary;\ \ read\ ())$$

as required.

Online resources

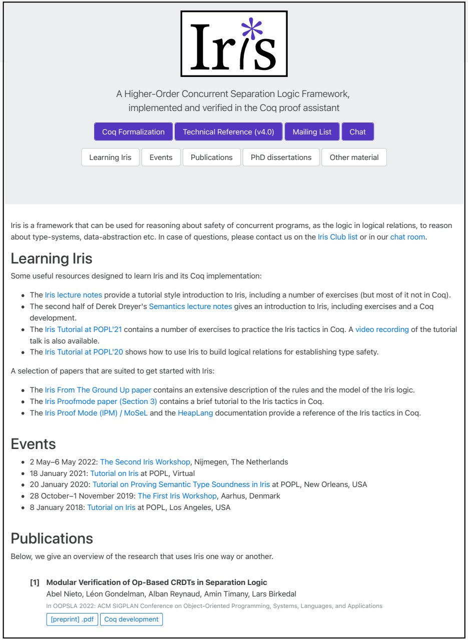
I hope this talk has made you interested in learning more about Iris, separation logic, program verification, *etc.*

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/leuvenss22/

- These slides
- Links to other resources


Iris

A Higher-Order Concurrent Separation Logic Framework, implemented and verified in the Coq proof assistant

Coq Formalization | Technical Reference (v4.0) | 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 second half of Derek Dreyer's Semantics lecture notes gives an introduction to Iris, including exercises and a Coq development.
- The Iris Tutorial at POPL'21 contains a number of exercises to practice the Iris tactics in Coq. A video recording of the tutorial talk is also available.
- 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

- 2 May–6 May 2022: The Second Iris Workshop, Nijmegen, The Netherlands
- 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] **Modular Verification of Op-Based CRDTs in Separation Logic**
Abel Nieto, Léon Gondelman, Alban Reynaud, Amin Timany, Lars Birkedal
In OOPSLA 2022: ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications
[preprint] .pdf | Coq development