

# Foundational Verification of Concurrent and Distributed Programs

Amin Timany

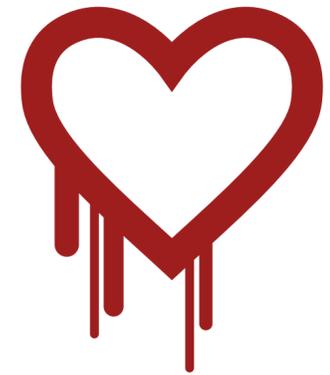
August 27<sup>th</sup>, 2025,

Programming Languages In Denmark (PLaID); Organized as Part of  
The 3<sup>rd</sup> Danish Digitalization, Data Science and AI (D3A 3.0) Conference,  
Nyborg, Denmark

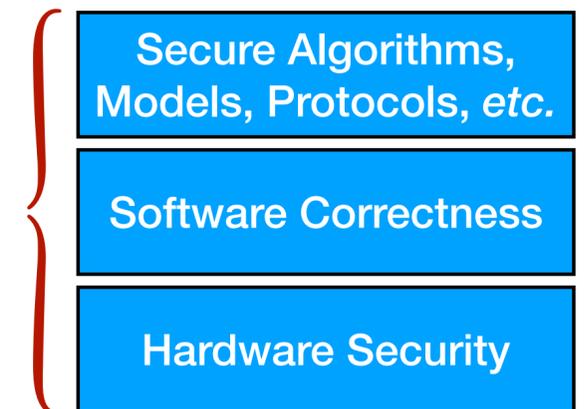
# Why Verify Programs?

## Software correctness is essential for security

- Bugs, compromising security can occur in implementations, even if the high-level models and protocols are correct.
  - Example: the infamous Heartbleed bug
    - A memory safety bug
- The entire stack should be secure
- February 2024: the White House issued a memorandum recommending memory safety and using formal methods



Formal and foundational techniques  
apply to the entire stack



# Why Verify Programs?

## Not just security: robustness of infrastructural software

- Bugs can also lead to
  - Data corruption, service unavailability, *etc.*
  - Can incur a hefty cost
- Example:
  - The CrowdStrike outage in July 2024
    - Affected the operation of airports, hospitals, *etc.*
    - Another memory safety bug

Blue screens of death at LGA airport from the CrowdStrike 2024 July outage

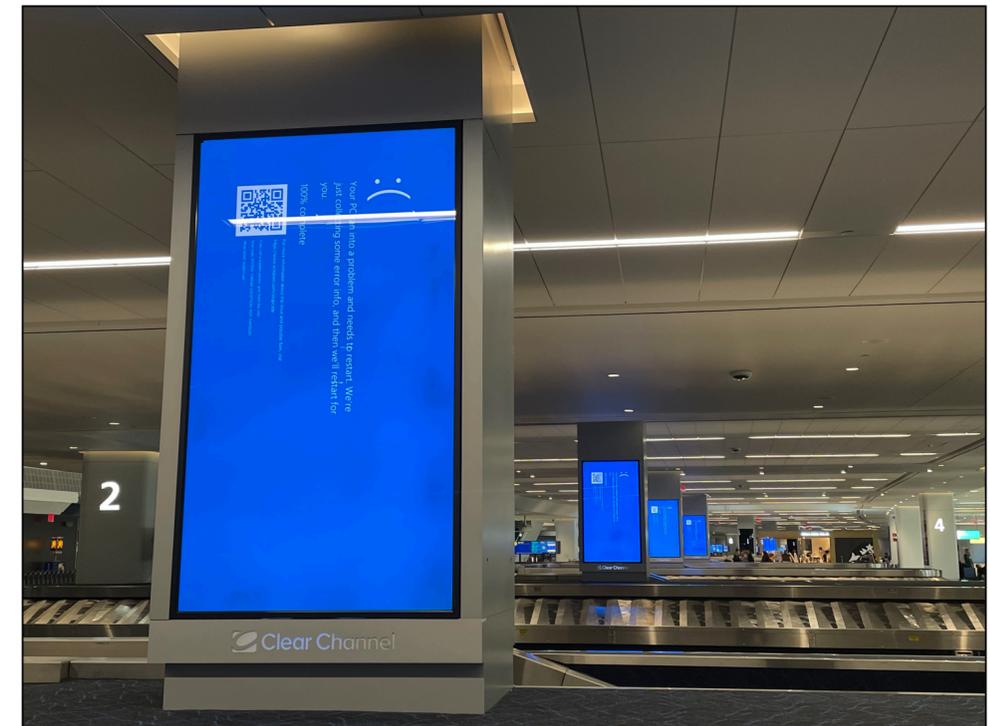
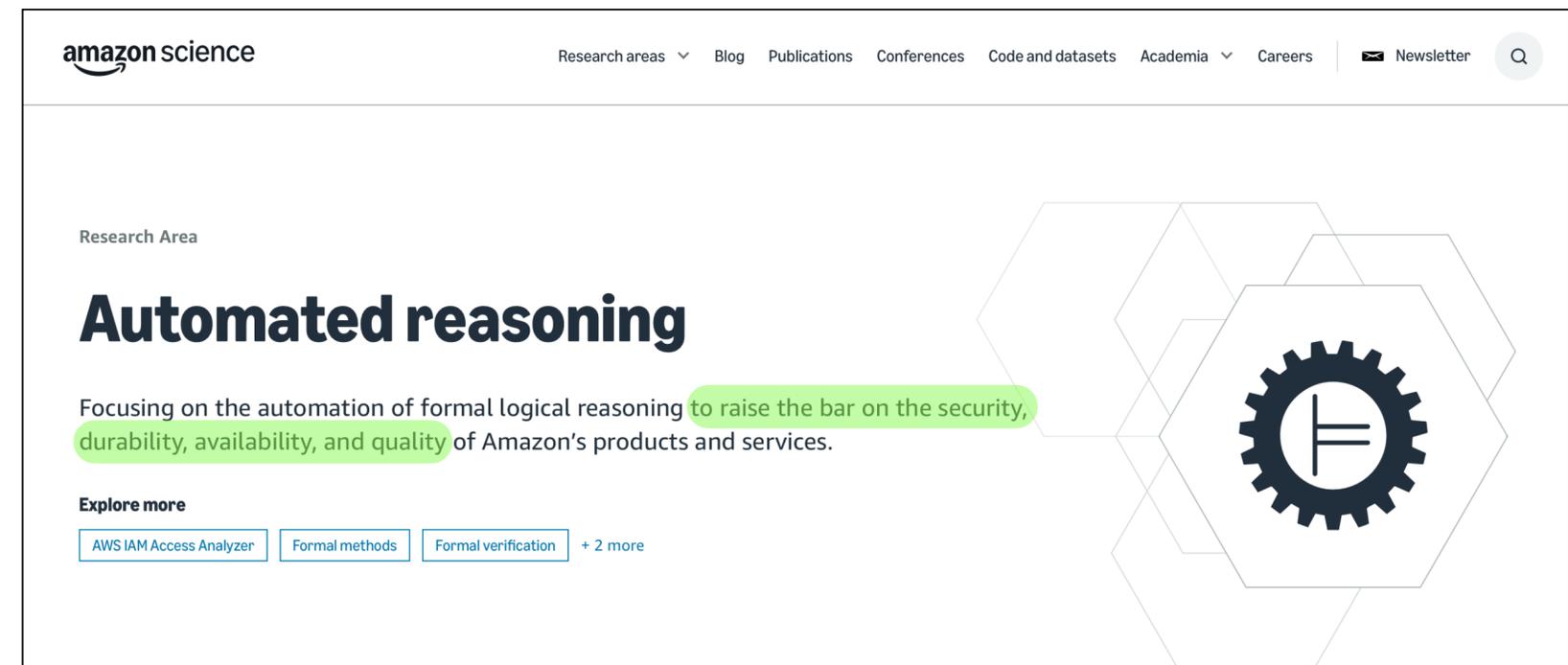


Photo by the user Smishra1 on wikimedia.org under the Creative Commons Attribution-Share Alike 4.0 International license.

# Why Verify Programs?

## Not just security: robustness of infrastructural software

- Some companies, notably AWS, have embraced formal reasoning
- The formal guarantees give them a competitive edge in the market



<https://www.amazon.science/research-areas/automated-reasoning>  
[accessed on Aug 4, 2025]

# Formal and Foundational Verification

- Prove correctness of programs (including memory safety)
- Formal and Foundational (from first principles)
  - We start by giving **semantics** (meaning) to programs
  - We develop and use **program logics** to prove correctness of programs

# Semantics & Safety Verification

# Operational Semantics

- We take a simple (OCaml-like) functional imperative language
- Define a relation “ $\rightarrow$ ” that describes **individual steps of computation**
  - Captures enough details to rule out (the class of) bugs we are interested in

# Operational Semantics

- We take a simple (OCaml-like) functional imperative language
- Define a relation “ $\rightarrow$ ” that describes **individual steps of computation**
  - Captures enough details to rule out (the class of) bugs we are interested in

$$2 + 3 \rightarrow 5$$

**Pronounced: “reduces to”, “steps to”, *etc.***

# Operational Semantics

- We take a simple (OCaml-like) functional imperative language
- Define a relation “ $\rightarrow$ ” that describes **individual steps of computation**
  - Captures enough details to rule out (the class of) bugs we are interested in

if not true then 2 else 3  $\rightarrow$  if false then 2 else 3  $\rightarrow$  3

# Operational Semantics

- We take a simple (OCaml-like) functional imperative language
- Define a relation “ $\rightarrow$ ” that describes **individual steps of computation**
  - Captures enough details to rule out (the class of) bugs we are interested in

$$\begin{aligned} & (\text{rec } f \ x \ := \ \text{if } x == 0 \ \text{then } 1 \ \text{else } x \times f \ (x - 1)) \ 5 \rightarrow \\ & \text{if } 5 == 0 \ \text{then } 1 \\ & \text{else } 5 \times (\text{rec } f \ x \ := \ \text{if } x == 0 \ \text{then } 1 \ \text{else } x \times f \ (x - 1)) \ (5 - 1) \end{aligned}$$

# Operational Semantics

- We take a simple (OCaml-like) functional imperative language
- Define a relation “ $\rightarrow$ ” that describes **individual steps of computation**
  - Captures enough details to rule out (the class of) bugs we are interested in

if "a" then 2 else 3  $\rightarrow$  ?

# Operational Semantics

- We take a simple (OCaml-like) functional imperative language
- Define a relation “ $\rightarrow$ ” that describes **individual steps of computation**
  - Captures enough details to rule out (the class of) bugs we are interested in

if "a" then 2 else 3  $\nrightarrow$

**Getting stuck indicates an error**



**Similarly for memory violations, e.g., array index out of bounds**

# Safety and Partial Functional Correctness

- **Safety:** the program does not crash (does not get stuck)

$$\text{Safe}(e) := \forall e'. e \rightarrow^* e' \implies \text{Val}(e') \vee \exists e''. e' \rightarrow e''$$

Zero or more steps

A value: any thing that can be passed as argument, e.g., a number, a pointer

# Safety and Partial Functional Correctness

- **Safety:** the program does not crash (does not get stuck)

$$\text{Safe}(e) := \forall e'. e \rightarrow^* e' \implies \text{Val}(e') \vee \exists e''. e' \rightarrow e''$$

Zero or more steps

A value: any thing that can be passed as argument, e.g., a number, a pointer

- Example:  $\text{Safe} \left( (\text{rec } f \ x := \text{if } x == 0 \text{ then } 1 \text{ else } x \times f(x - 1)) \ 5 \right)$

# Safety and Partial Functional Correctness

- **Safety:** the program does not crash (does not get stuck)

$$\text{Safe}(e) := \forall e'. e \rightarrow^* e' \implies \text{Val}(e') \vee \exists e''. e' \rightarrow e''$$

Zero or more steps

A value: any thing that can be passed as argument, e.g., a number, a pointer

- Example:  $\text{Safe} \left( (\text{rec } f \ x := \text{if } x == 0 \text{ then } 1 \text{ else } x \times f(x - 1)) \ 5 \right)$
- Counterexample:  $\neg \text{Safe} (\text{if } \text{"a"} \text{ then } 2 \text{ else } 3)$

# Safety and Partial Functional Correctness

- **Safety:** the program does not crash (does not get stuck)

$$\text{Safe}(e) := \forall e'. e \rightarrow^* e' \implies \text{Val}(e') \vee \exists e''. e' \rightarrow e''$$

Zero or more steps

A value: any thing that can be passed as argument, e.g., a number, a pointer

- Example:  $\text{Safe} \left( (\text{rec } f \ x := \text{if } x == 0 \text{ then } 1 \text{ else } x \times f(x - 1)) \ 5 \right)$
- Counterexample:  $\neg \text{Safe} (\text{if } "a" \text{ then } 2 \text{ else } 3)$
- **Q:** How about this program:  $(\text{fun } x := \text{if } "a" \text{ then } x \text{ else } x + 1)$ ? Is it safe?
  - **Q:** Should functions be considered values?

# Safety and Partial Functional Correctness

- **Safety:** the program does not crash (does not get stuck)

$$\text{Safe}(e) := \forall e'. e \rightarrow^* e' \implies \text{Val}(e') \vee \exists e''. e' \rightarrow e''$$

Zero or more steps

A value: any thing that can be passed as argument, e.g., a number, a pointer

- Example:  $\text{Safe} \left( (\text{rec } f \ x := \text{if } x == 0 \text{ then } 1 \text{ else } x \times f(x - 1)) \ 5 \right)$
- Counterexample:  $\neg \text{Safe} (\text{if } "a" \text{ then } 2 \text{ else } 3)$
- **Q:** How about this program:  $(\text{fun } x := \text{if } "a" \text{ then } x \text{ else } x + 1)$ ? Is it safe?
  - **Q:** Should functions be considered values?
- How about this other program:  $(\text{fun } x := \text{if } "a" \text{ then } x \text{ else } x + 1) \ 5$ ?

# Safety and Partial Functional Correctness

- **Partial Functional correctness:** safe & upon termination the postcondition  $\phi$  holds

$$\text{Correct}_{\phi}(e) := \forall e'. e \rightarrow^* e' \implies (\text{Val}(e') \wedge \phi(e')) \vee \exists e''. e' \rightarrow e''$$

**The postcondition**



# Safety and Partial Functional Correctness

- **Partial Functional correctness:** safe & upon termination the postcondition  $\phi$  holds

$$\text{Correct}_{\phi}(e) := \forall e'. e \rightarrow^* e' \implies (\text{Val}(e') \wedge \phi(e')) \vee \exists e''. e' \rightarrow e''$$

**The postcondition**



- Example:  $\text{Correct}_{isOdd}(2 + 3)$

# Safety and Partial Functional Correctness

- **Partial Functional correctness:** safe & upon termination the postcondition  $\phi$  holds

$$\text{Correct}_{\phi}(e) := \forall e'. e \rightarrow^* e' \implies (\text{Val}(e') \wedge \phi(e')) \vee \exists e''. e' \rightarrow e''$$

**The postcondition**



- Example:  $\text{Correct}_{\text{isOdd}}(2 + 3)$
- **Q:** What should  $\phi$  be for  $\text{Correct}_{\phi}(\text{rec } f \ x := \text{if } x == 0 \text{ then } 1 \text{ else } x \times f(x - 1))$ ?

# Safety and Partial Functional Correctness

- **Partial Functional correctness:** safe & upon termination the postcondition  $\phi$  holds

$$\text{Correct}_{\phi}(e) := \forall e'. e \rightarrow^* e' \implies (\text{Val}(e') \wedge \phi(e')) \vee \exists e''. e' \rightarrow e''$$

**The postcondition**



- Example:  $\text{Correct}_{\text{isOdd}}(2 + 3)$
- **Q:** What should  $\phi$  be for  $\text{Correct}_{\phi}(\text{rec } f \ x := \text{if } x == 0 \text{ then } 1 \text{ else } x \times f(x - 1))$ ?

$$\phi(v) := \forall n \in \mathbb{Z}. n \geq 0 \implies \text{Correct}_{\in \mathbb{Z}}(v \ n)$$

**Could be stronger, e.g., the result must be  $n$  factorial**



# Safety and Partial Functional Correctness

- How do we prove safety/functional correctness?
- Reasoning directly in terms of operational semantics does not scale!

# Safety and Partial Functional Correctness

- How do we prove safety/functional correctness?
- Reasoning directly in terms of operational semantics does not scale!
  - For concurrent programs we need to consider all possible thread interleavings
    - Weak memory significantly complicates things further

# Safety and Partial Functional Correctness

- How do we prove safety/functional correctness?
- Reasoning directly in terms of operational semantics does not scale!
  - For concurrent programs we need to consider all possible thread interleavings
    - Weak memory significantly complicates things further
  - For distributed systems we need to consider all network behavior

# Safety and Partial Functional Correctness

- How do we prove safety/functional correctness?
- Reasoning directly in terms of operational semantics does not scale!
  - For concurrent programs we need to consider all possible thread interleavings
    - Weak memory significantly complicates things further
  - For distributed systems we need to consider all network behavior
  - How do we even specify correctness of memory accesses?
    - Is the following program safe? `if ! $\ell < 0$  then 0 else ! $\ell + 1$`

**Loading from memory location  $\ell$**



# Safety and Partial Functional Correctness

- How do we prove safety/functional correctness?
- Reasoning directly in terms of operational semantics does not scale!
  - For concurrent programs we need to consider all possible thread interleavings
    - Weak memory significantly complicates things further
  - For distributed systems we need to consider all network behavior
  - How do we even specify correctness of memory accesses?
    - Is the following program safe? `if ! $\ell < 0$  then 0 else ! $\ell + 1$`

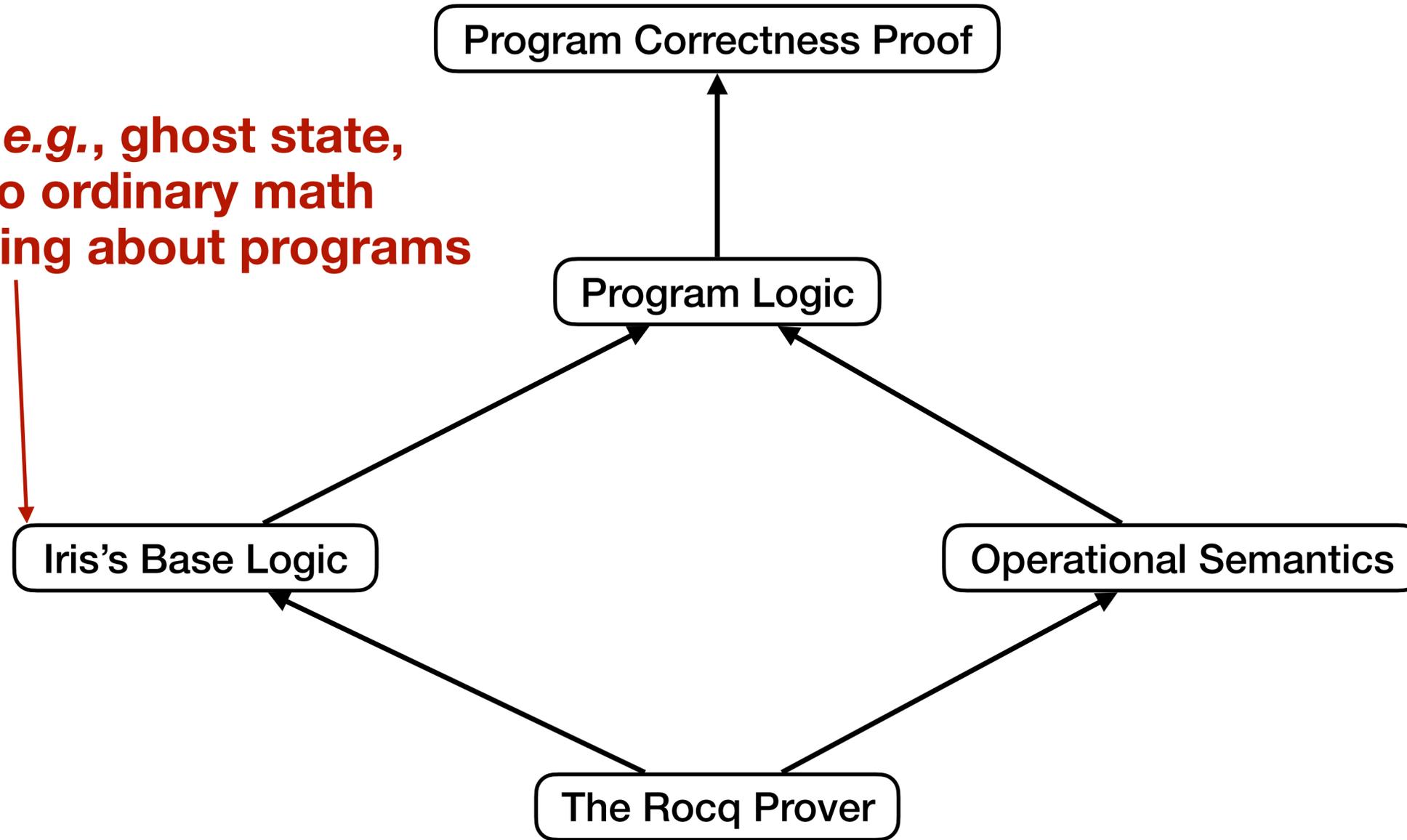
Loading from memory location  $\ell$

**The rest of the talk: how program logics address these issues**

# Program Logics: the Iris Framework

# The Iris Framework

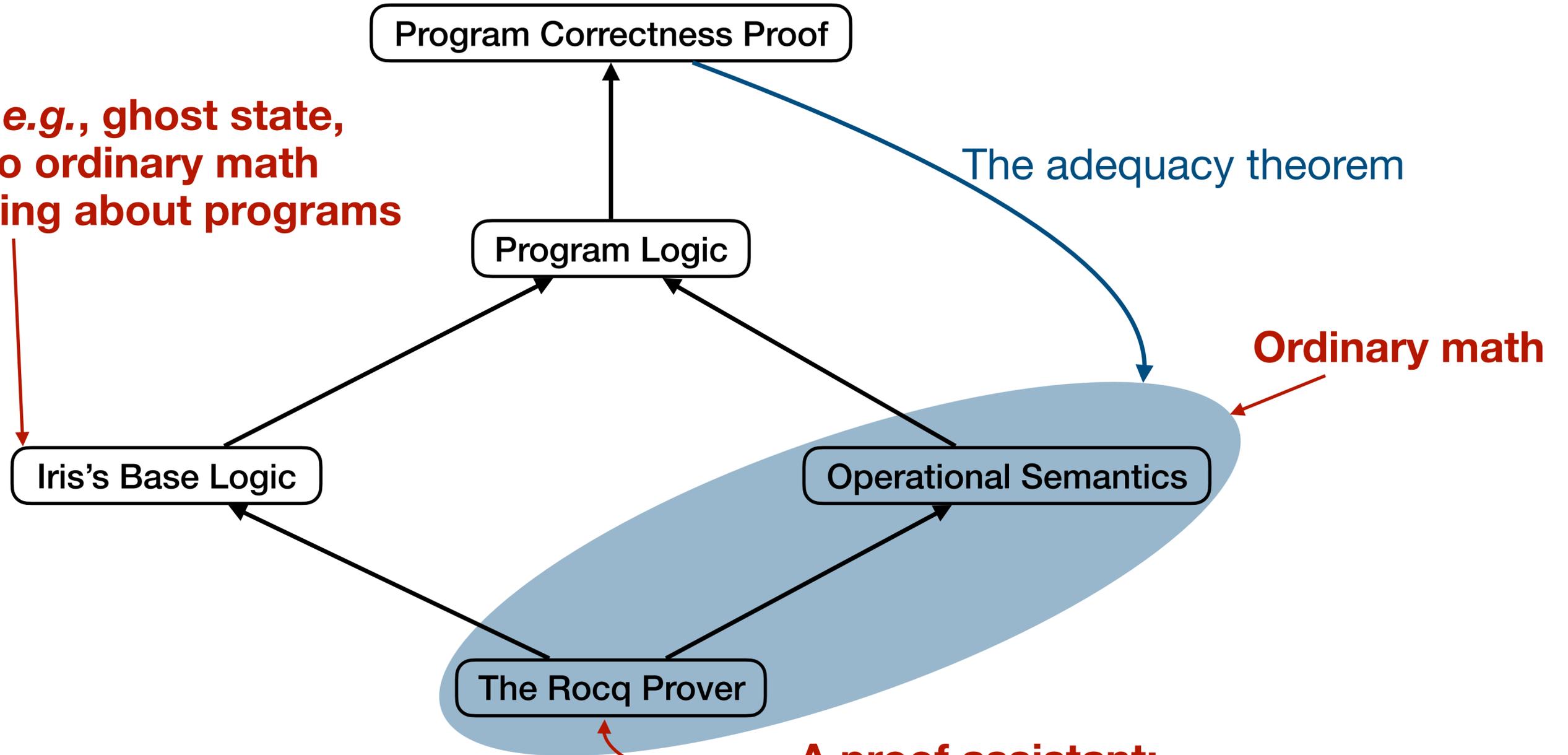
Logical primitives, e.g., ghost state, invariants, added to ordinary math to facilitate reasoning about programs



**A proof assistant:  
A tool to develop mathematical theories  
and proofs checked by the machine**

# The Iris Framework

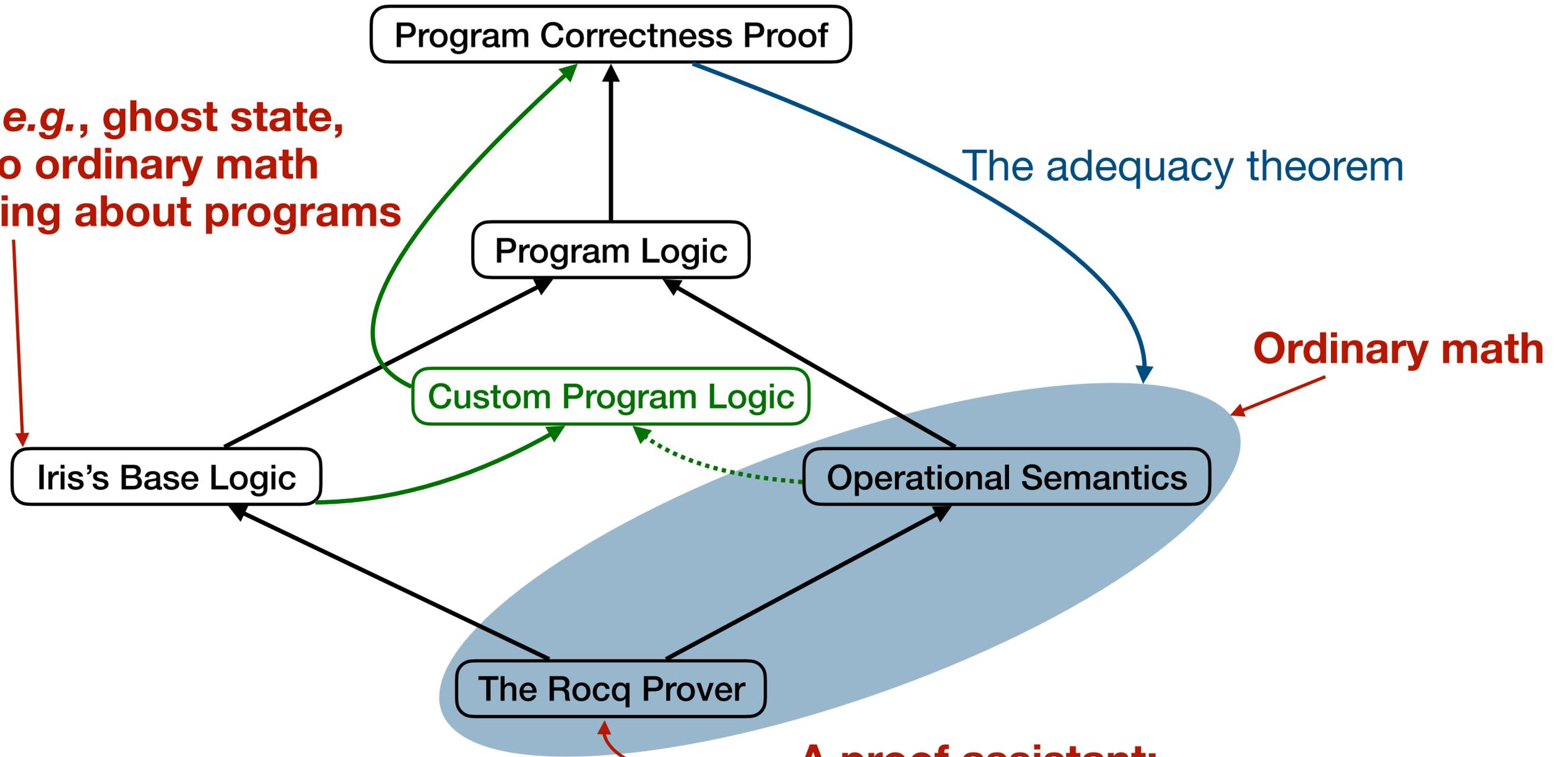
Logical primitives, e.g., ghost state, invariants, added to ordinary math to facilitate reasoning about programs



**A proof assistant:  
A tool to develop mathematical theories  
and proofs checked by the machine**

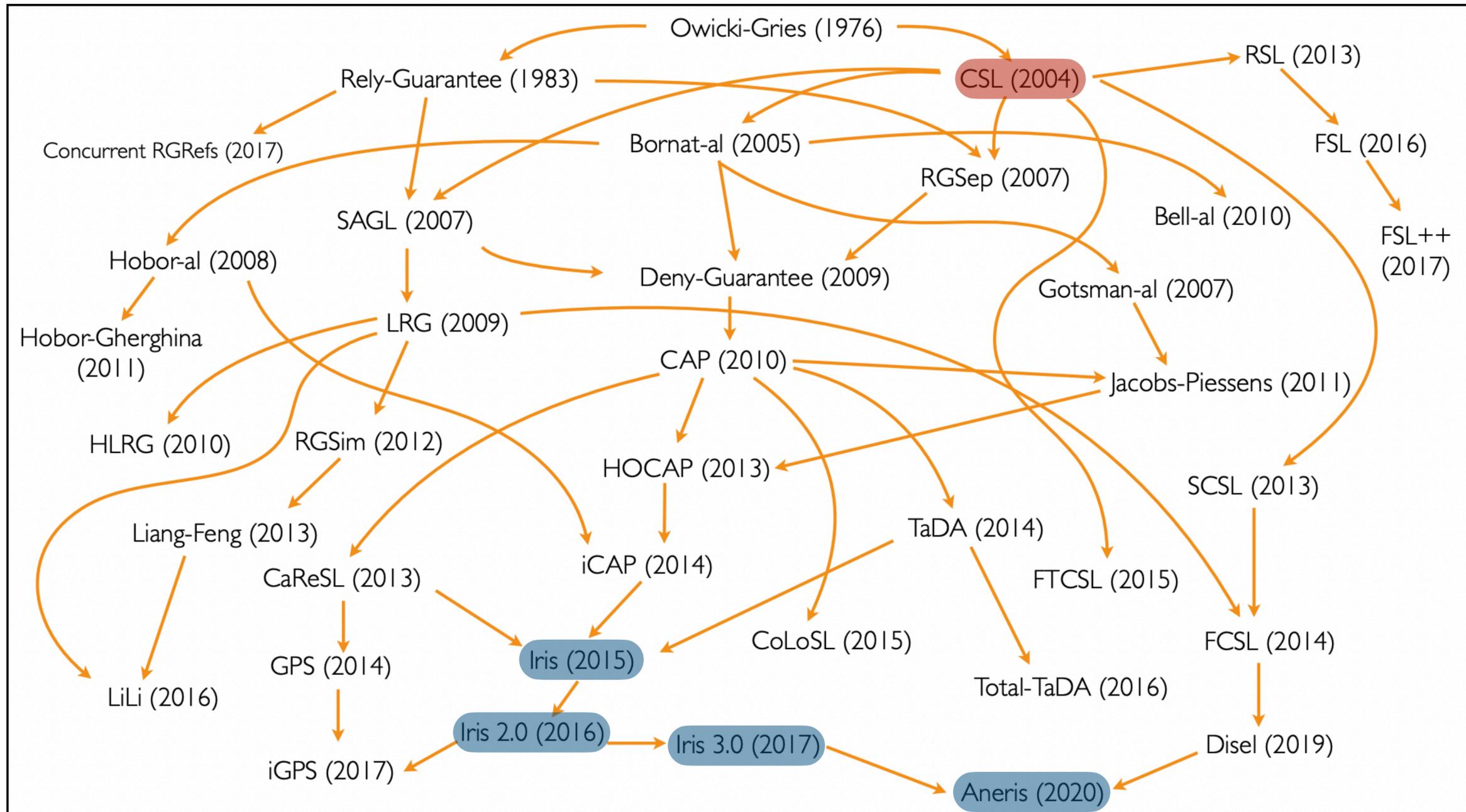
# The Iris Framework

Logical primitives, e.g., ghost state, invariants, added to ordinary math to facilitate reasoning about programs



**A proof assistant:**  
A tool to develop mathematical theories and proofs checked by the machine

# Program Logics for Concurrent and Distributed Programs



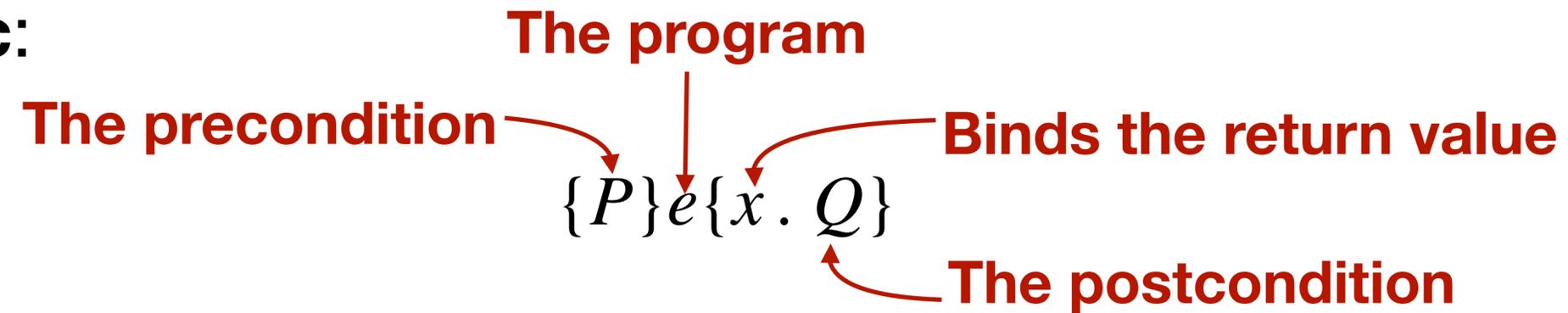
Taken from [wikimedia.org](https://commons.wikimedia.org/wiki/File:Flow_of_ideas_in_program_logics_for_concurrent_and_distributed_programs.png), by Ilya Sergey

Description by the author: This image depicts the "flow of ideas" that have been implemented in various logical frameworks for proving correctness of concurrent and distributed programs.

# Iris's Program Logic

## A Higher-Order Separation Logic

A Hoare-style logic:



Examples:

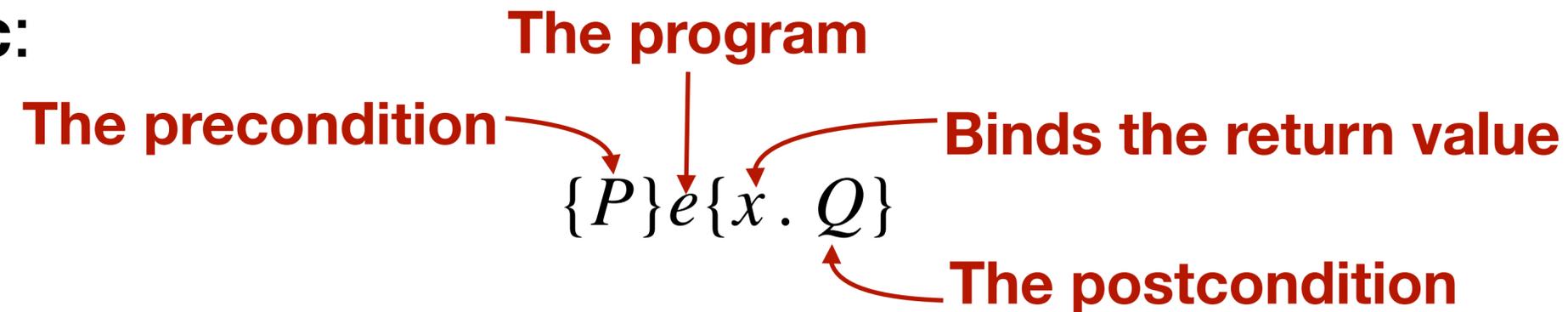
$\{\text{True}\}2 + 3\{x. \text{isOdd}(x)\}$

$\{?\}(\text{rec } f\ x := \text{if } x == 0 \text{ then } 1 \text{ else } x \times f(x - 1))\ n\{x. ?\}$

# Iris's Program Logic

## A Higher-Order Separation Logic

A Hoare-style logic:



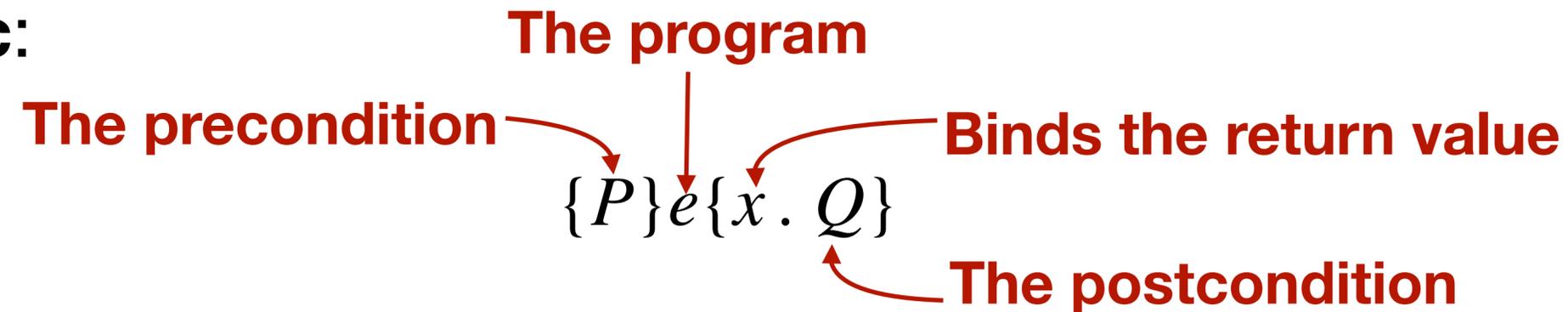
Examples:

$$\{\text{True}\}2 + 3\{x. \text{isOdd}(x)\}$$
$$\{n \geq 0\}(\text{rec } f \ x := \text{if } x == 0 \text{ then } 1 \text{ else } x \times f(x - 1)) \ n\{x. x = n!\}$$

# Iris's Program Logic

## A Higher-Order Separation Logic

A Hoare-style logic:



Examples:

$$\{\text{True}\}2 + 3\{x. \text{isOdd}(x)\}$$
$$\{n \geq 0\}(\text{rec } f \ x := \text{if } x == 0 \text{ then } 1 \text{ else } x \times f(x - 1)) \ n\{x. x = n!\}$$

**The program logic provides an *expressive language* to specify correctness of programs, and versatile *reasoning principles* to carry out proofs**

# The Adequacy Theorem

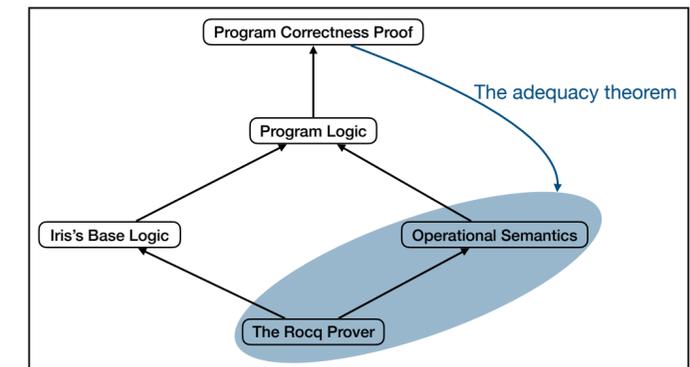
$$\{\text{True}\}2 + 3\{x. \text{isOdd}(x)\}$$

## Theorem (Adequacy)

*If we prove*

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

*in the program logic of Iris, then  $\text{Correct}_\phi(e)$  holds.*



Recall the intuition of Adequacy:  
if a program is proven correct  
inside the program logic, then it is  
correct.

# Separation Logic

**Separating Conjunction:**

$P \star Q$   **the separating conjunction**

$P \star Q$  holds if both  $P$  and  $Q$  hold, but for *disjoint* resources

# Separation Logic

**Separating Conjunction:**

$P \star Q$   **the separating conjunction**

$P \star Q$  holds if both  $P$  and  $Q$  hold, but for *disjoint* resources

**Example:** exclusive ownership of a memory location

$\ell \mapsto v$   **the points-to proposition**

# Separation Logic

## Separating Conjunction:

$P \star Q$   the separating conjunction

$P \star Q$  holds if both  $P$  and  $Q$  hold, but for *disjoint* resources

## Example: exclusive ownership of a memory location

$\ell \mapsto v$   the points-to proposition

### Points-to-Exclusive

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

### Hoare-Alloc

$\{\text{True}\} \text{ref } v \{ \ell . \ell \mapsto v \}$

### Hoare-Store

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

### Hoare-Load

$\{ \ell \mapsto v \} !\ell \{ x . x = v \star \ell \mapsto v \}$

# Concurrent Separation Logic

## Disjoint Concurrency

Threads working on disjoint resources can be safely run in parallel

**Hoare-Par**

$$\frac{\{P_1\}e_1\{x. Q_1(x)\} \quad \{P_2\}e_2\{x. Q_2(x)\}}{\{P_1 \star P_2\}e_1 \parallel e_2\{x. \exists v_1, v_2. x = (v_1, v_2) \star Q_1(v_1) \star Q_2(v_2)\}}$$

# Concurrent Separation Logic

## Disjoint Concurrency

Threads working on disjoint resources can be safely run in parallel

**Hoare-Par**

$$\frac{\{P_1\}e_1\{x. Q_1(x)\} \quad \{P_2\}e_2\{x. Q_2(x)\}}{\{P_1 \star P_2\}e_1 \parallel e_2\{x. \exists v_1, v_2. x = (v_1, v_2) \star Q_1(v_1) \star Q_2(v_2)\}}$$

**Q:** What if the two threads share resources?

# Concurrent Separation Logic

## Disjoint Concurrency

Threads working on disjoint resources can be safely run in parallel

**Hoare-Par**

$$\frac{\{P_1\}e_1\{x. Q_1(x)\} \quad \{P_2\}e_2\{x. Q_2(x)\}}{\{P_1 \star P_2\}e_1 \parallel e_2\{x. \exists v_1, v_2. x = (v_1, v_2) \star Q_1(v_1) \star Q_2(v_2)\}}$$

**Q:** What if the two threads share resources?

We use an **invariant**  $\boxed{P}$  which asserts that  $P$  always holds, and all threads respect it.

**Inv-Duplicable**

$$\boxed{P} \dashv\vdash \boxed{P} \star \boxed{P}$$

**Inv-Access**

$$\frac{\{P \star I\}e\{x. Q \star I\} \quad e \text{ is atomic}}{\{P \star \boxed{I}\}e\{x. Q\}}$$

# Concurrent Separation Logic

## Shared Memory Concurrency

Consider the following program

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

# Concurrent Separation Logic

## Shared Memory Concurrency

Consider the following program

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

It must return a non-negative result. **Q:** how do we specify that?

# Concurrent Separation Logic

## Shared Memory Concurrency

Consider the following program

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

It must return a non-negative result. **Q:** how do we specify that?

**Q:** How do we prove that? **NB: This is non-disjoint concurrency!**

# Concurrent Separation Logic

## Shared Memory Concurrency

Consider the following program

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

It must return a non-negative result. **Q:** how do we specify that?

**Q:** How do we prove that? **NB: This is non-disjoint concurrency!**

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

# Concurrent Separation Logic

## Shared Memory Concurrency: “The Proof”

{True}

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

{ $c \mapsto 0$ }

{ $\exists n. c \mapsto n \star n \geq 0$ }

$$\left( \begin{array}{l|l} \{ \exists n. c \mapsto n \star n \geq 0 \} & \{ \exists n. c \mapsto n \star n \geq 0 \} \\ \{ c \mapsto k \star k \geq 0 \} & \{ c \mapsto j \star j \geq 0 \} \\ \text{faa } c \ 1 & \text{faa } c \ 2 \\ \{ c \mapsto k + 1 \star k \geq 0 \} & \{ c \mapsto j + 2 \star j \geq 0 \} \\ \{ \exists n. c \mapsto n \star n \geq 0 \} & \{ \exists n. c \mapsto n \star n \geq 0 \} \end{array} \right);$$

{ $\exists n. c \mapsto n \star n \geq 0$ }

!c

{ $x. x \geq 0$ }

# Concurrent Separation Logic

## Shared Memory Concurrency: A Stronger Postcondition

What if we wish to prove something stronger?

```
{True}
let c = ref 0 in
  (faa c 1 || faa c 2);
!c
{x . x = 3}
```

**Q:** What invariant should we take now?

# Concurrent Separation Logic

## Shared Memory Concurrency: A Stronger Postcondition

What if we wish to prove something stronger?

```
{True}  
let c = ref 0 in  
  (faa c 1 || faa c 2);  
!c  
{x . x = 3}
```

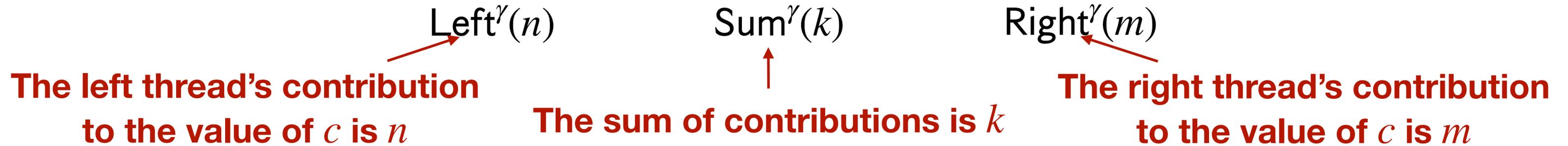
**Q:** What invariant should we take now?

**Not possible with invariants alone!**  
**We need *ghost state* to keep account of each thread's contribution.**

# Concurrent Separation Logic

## Ghost State

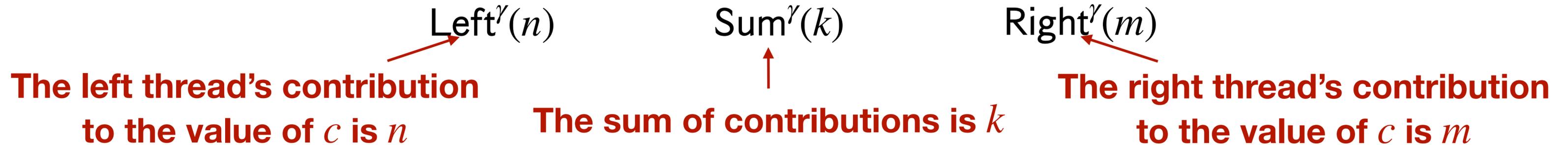
We use user-defined resources to define the following propositions in the logic:



# Concurrent Separation Logic

## Ghost State

We use user-defined resources to define the following propositions in the logic:



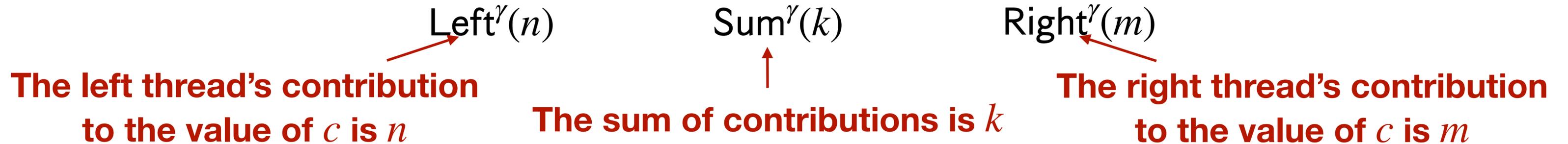
For this stronger property, after allocating  $c$  we establish the following:

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

# Concurrent Separation Logic

## Ghost State

We use user-defined resources to define the following propositions in the logic:



For this stronger property, after allocating  $c$  we establish the following:

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

And, after the two threads are run:

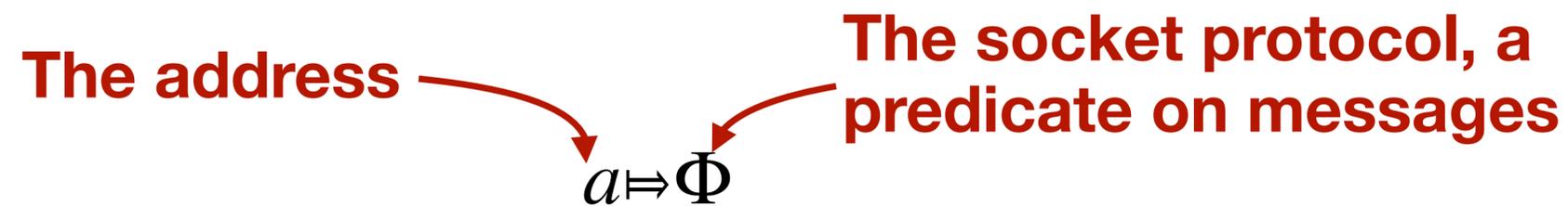
$$\{ \text{Left}^\gamma(1) \star \text{Right}^\gamma(2) \star \boxed{\exists n. c \mapsto n \star \text{Sum}^\gamma(n)} \}$$

# Verification of Distributed Systems

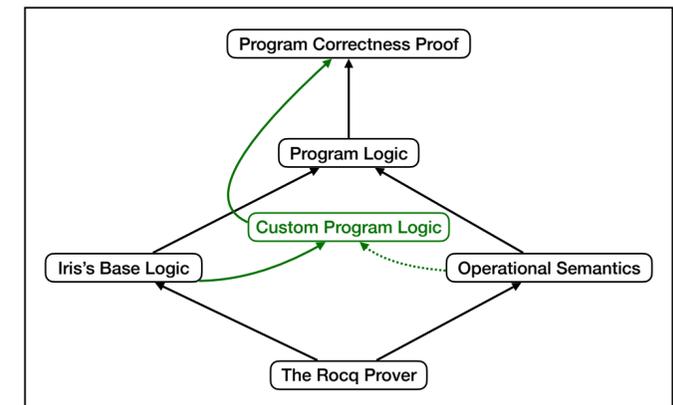
# Modular Verification of Distributed Systems

## The Aneris Program Logic

- The key to Aneris’s modularity (in addition to Hoare triples, separation logic, *etc.*):
  - so-called “socket protocols”



- Sending message  $m$  to address  $a$ : we should prove  $\Phi(m)$
- Receiving message  $m$  on a socket bound to  $a$ : we know  $\Phi(m)$



Aneris is one such custom program logic

# Modular Verification of Distributed Systems

## Applications of the Aneris Program Logic

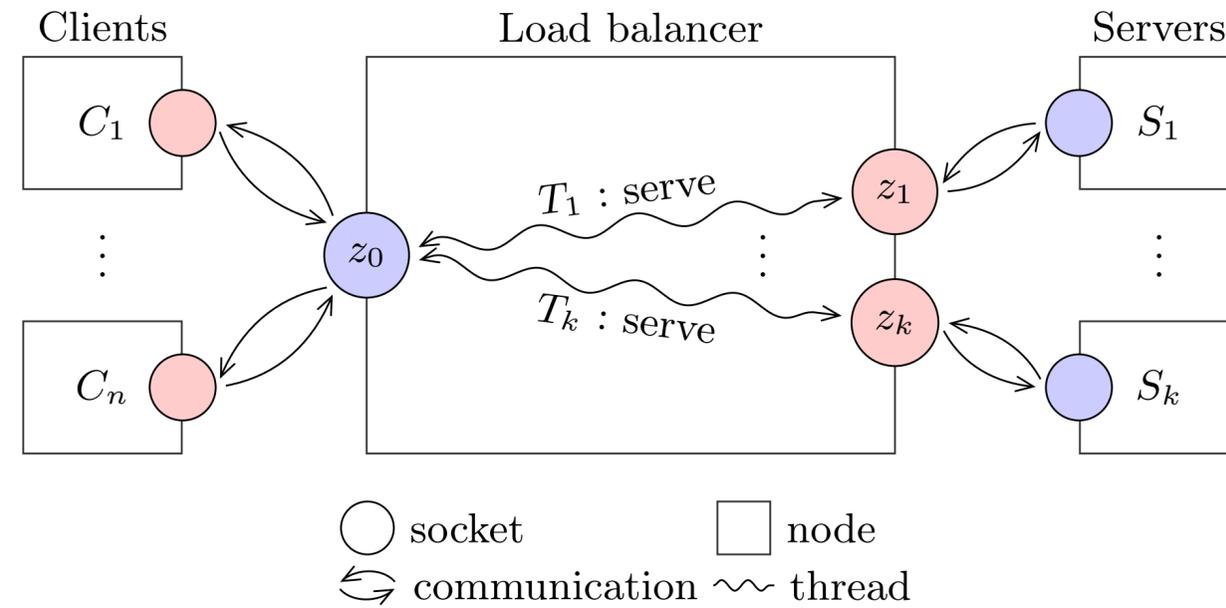
We have used Aneris to verify of the following distributed systems and their clients

- **A Load balancer** — distributing clients' load among multiple servers
- Causally consistent distributed key-value store
- Two-phase commit — coordinating transactions among multiple servers
- Conflict-free replicated data types (CRDTs)
- Reliable communication on top of an unreliable UDP-like network specified using session types
- Single-Decree Paxos — well known distributed consensus algorithm
- Correctness of a database implementing snapshot isolation

# Verification of Distributed Systems

## The Aneris Program Logic

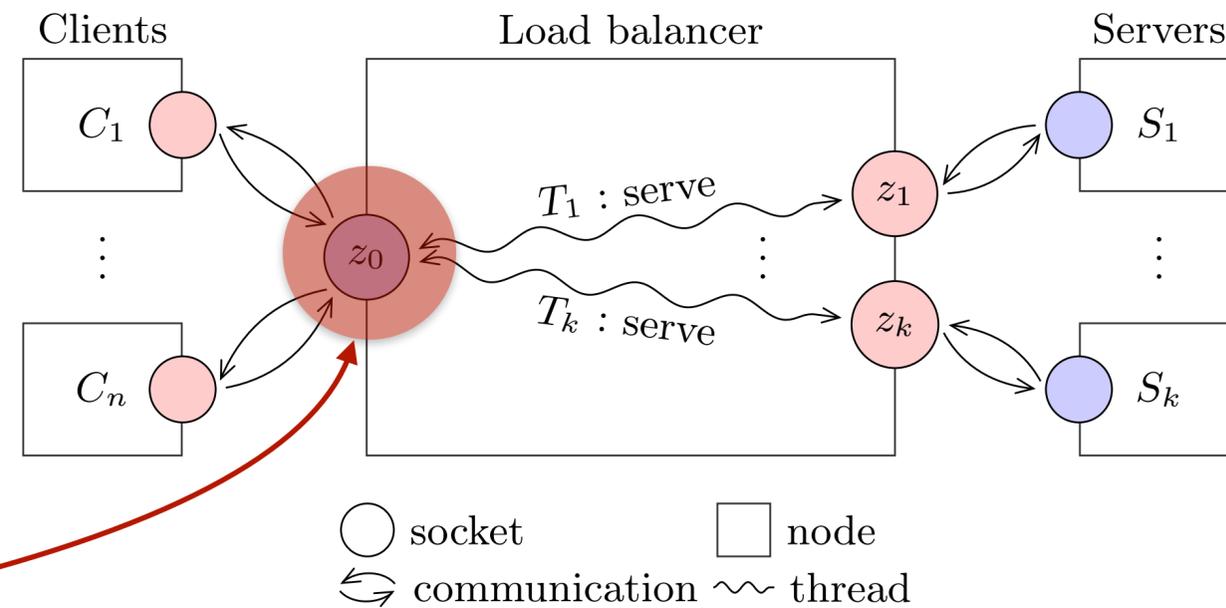
- A verified load balancer



# Verification of Distributed Systems

## The Aneris Program Logic

- A verified load balancer



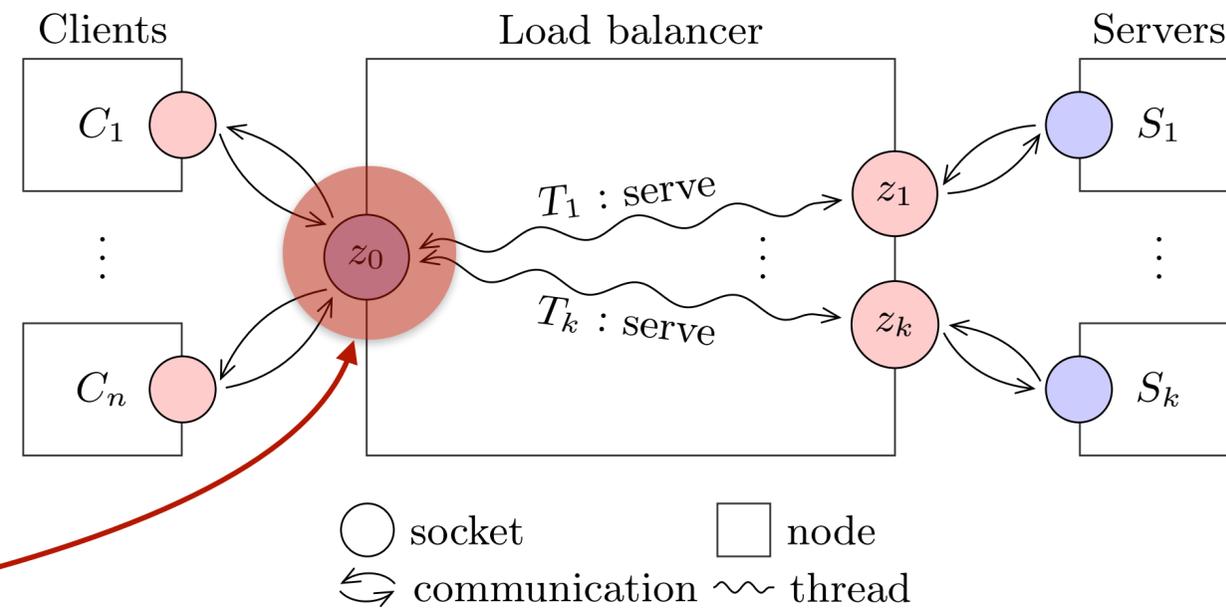
**What should the socket protocol be?**

- **Q:** What is a good specification for a general-purpose load balancer?

# Verification of Distributed Systems

## The Aneris Program Logic

- A verified load balancer



### What should the socket protocol be?

- **Q:** What is a good specification for a general-purpose load balancer?
  - Should act exactly as the server does

# Modular Verification of Distributed Systems

## The Aneris Program Logic

How do we state this formally?

Accepts any message  $m$  accepted by the server behind the load balancer

As long as the sender's socket protocol  $\Psi$  accepts server's response

$$\Phi_{LB}(m) = \Phi_{SRV}(m) \star \exists \Psi . \text{sender}(m) \Rightarrow \Psi \star \forall m' . \text{server\_response}(m, m') \implies \Psi(m')$$

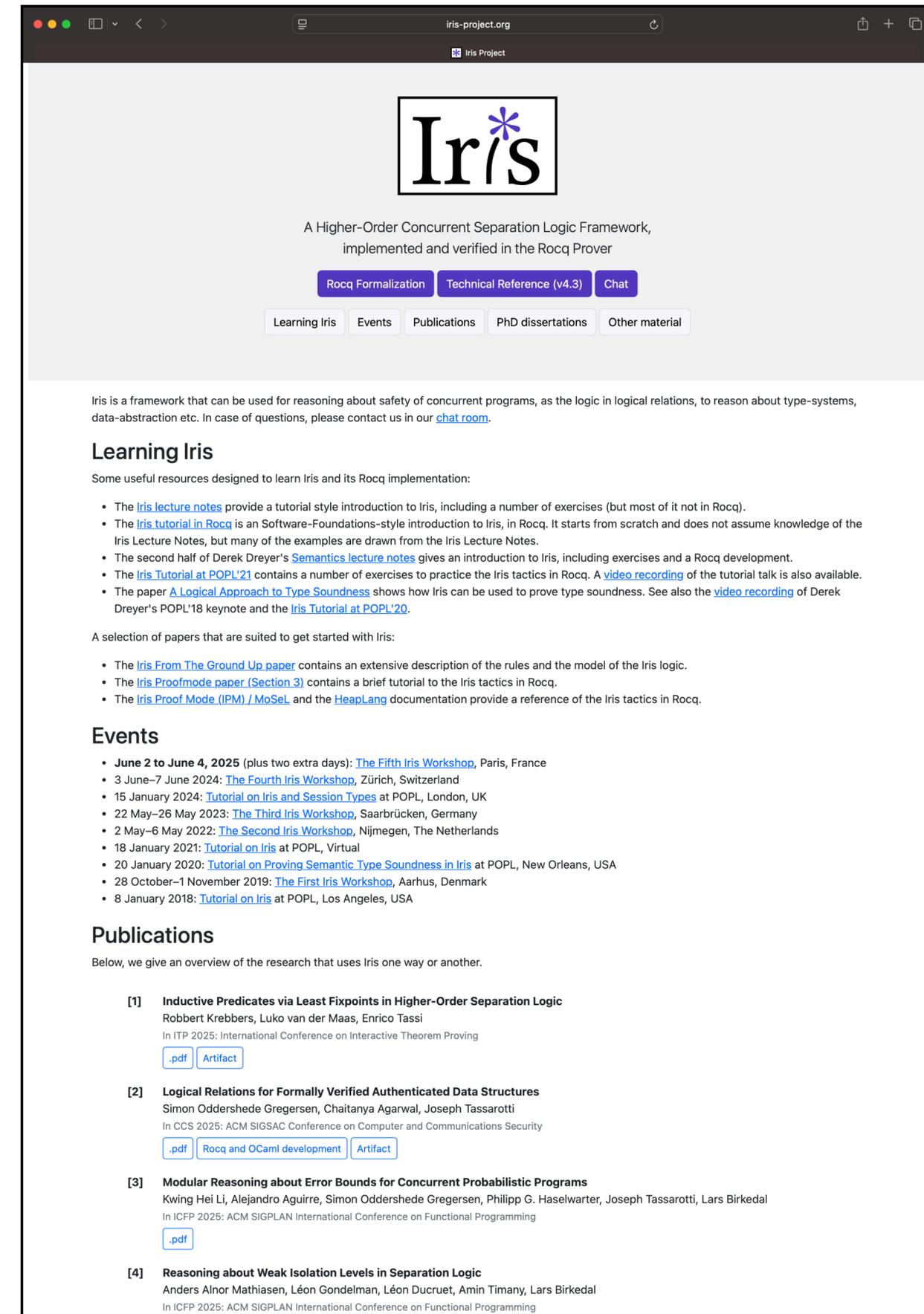
Higher-order: quantifies over protocols to define a protocol

Necessary for modularity: no need to specify upfront who can contact a node

# The Iris Framework: A Tour De Force

# Iris Has Also Been Used to

- Proving semantic type soundness of type systems including Scala, Rust, and Wasm
- Proving (contextual) equivalence of programs
- Proving time and space bound of algorithms
- Proving robust safety (safe interaction with unverified code)
  - Both at assembly level and high-level interactions
- See <https://iris-project.org>
  - 135 publications, 21 PhD theses



The screenshot shows the Iris Project website. At the top, there is a navigation menu with links for "Rocq Formalization", "Technical Reference (v4.3)", "Chat", "Learning Iris", "Events", "Publications", "PhD dissertations", and "Other material". Below the navigation, the website describes Iris as a Higher-Order Concurrent Separation Logic Framework, implemented and verified in the Rocq Prover. It then provides a list of resources for learning Iris, including lecture notes, tutorials, and papers. The "Events" section lists several workshops and tutorials, and the "Publications" section lists four recent papers with links to their PDFs and artifacts.

Iris Project

A Higher-Order Concurrent Separation Logic Framework,  
implemented and verified in the Rocq Prover

Rocq Formalization Technical Reference (v4.3) 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 in our [chat room](#).

### Learning Iris

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

- The [Iris lecture notes](#) provide a tutorial style introduction to Iris, including a number of exercises (but most of it not in Rocq).
- The [Iris tutorial in Rocq](#) is an Software-Foundations-style introduction to Iris, in Rocq. It starts from scratch and does not assume knowledge of the Iris Lecture Notes, but many of the examples are drawn from the Iris Lecture Notes.
- The second half of Derek Dreyer's [Semantics lecture notes](#) gives an introduction to Iris, including exercises and a Rocq development.
- The [Iris Tutorial at POPL'21](#) contains a number of exercises to practice the Iris tactics in Rocq. A [video recording](#) of the tutorial talk is also available.
- The paper [A Logical Approach to Type Soundness](#) shows how Iris can be used to prove type soundness. See also the [video recording](#) of Derek Dreyer's POPL'18 keynote and the [Iris Tutorial at POPL'20](#).

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 Rocq.
- The [Iris Proof Mode \(IPM\) / MoSeL](#) and the [HeapLang](#) documentation provide a reference of the Iris tactics in Rocq.

### Events

- **June 2 to June 4, 2025** (plus two extra days): [The Fifth Iris Workshop](#), Paris, France
- 3 June–7 June 2024: [The Fourth Iris Workshop](#), Zürich, Switzerland
- 15 January 2024: [Tutorial on Iris and Session Types](#) at POPL, London, UK
- 22 May–26 May 2023: [The Third Iris Workshop](#), Saarbrücken, Germany
- 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] Inductive Predicates via Least Fixpoints in Higher-Order Separation Logic**  
Robbert Krebbers, Luko van der Maas, Enrico Tassi  
In ITP 2025: International Conference on Interactive Theorem Proving  
[.pdf](#) [Artifact](#)
- [2] Logical Relations for Formally Verified Authenticated Data Structures**  
Simon Oddershede Gregersen, Chaitanya Agarwal, Joseph Tassarotti  
In CCS 2025: ACM SIGSAC Conference on Computer and Communications Security  
[.pdf](#) [Rocq and OCaml development](#) [Artifact](#)
- [3] Modular Reasoning about Error Bounds for Concurrent Probabilistic Programs**  
Kwing Hei Li, Alejandro Aguirre, Simon Oddershede Gregersen, Philipp G. Haselwarter, Joseph Tassarotti, Lars Birkedal  
In ICFP 2025: ACM SIGPLAN International Conference on Functional Programming  
[.pdf](#)
- [4] Reasoning about Weak Isolation Levels in Separation Logic**  
Anders Alnor Mathiasen, Léon Gondelman, Léon Ducruet, Amin Timany, Lars Birkedal  
In ICFP 2025: ACM SIGPLAN International Conference on Functional Programming

**Thanks!**