Foundational Verification of Concurrent and Distributed Programs

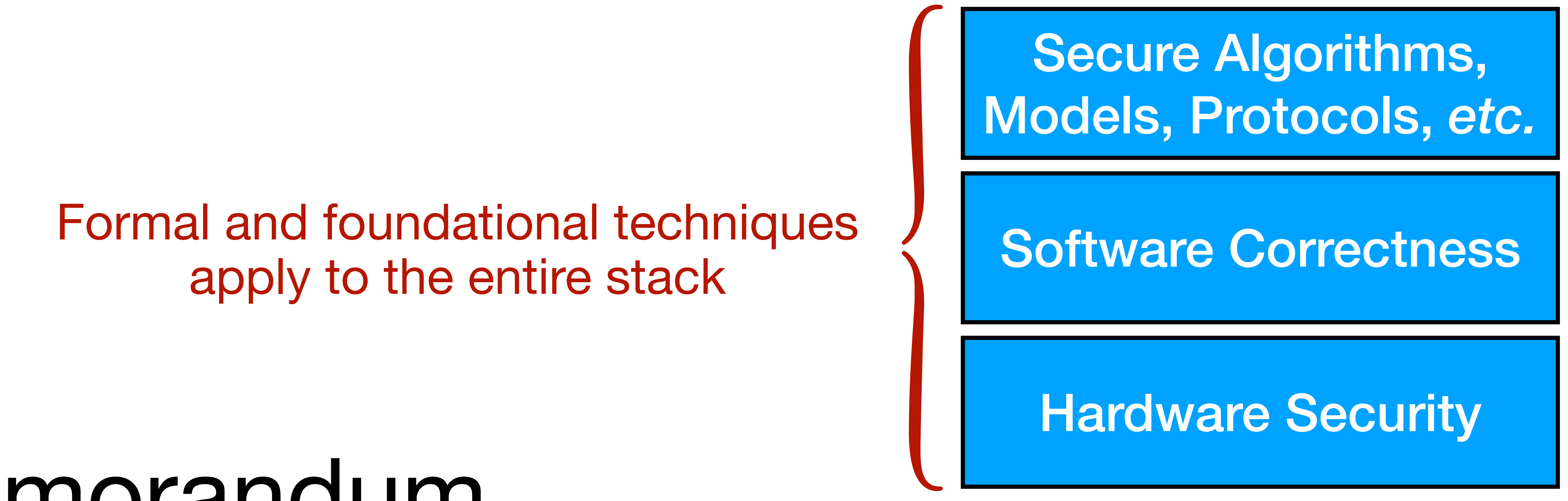
Amin Timany

August 27th, 2025,
Programming Languages In Denmark (PLaID); Organized as Part of
The 3rd 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 Hearbleed bug
        - A memory safety bug
- The entire stack should be secure



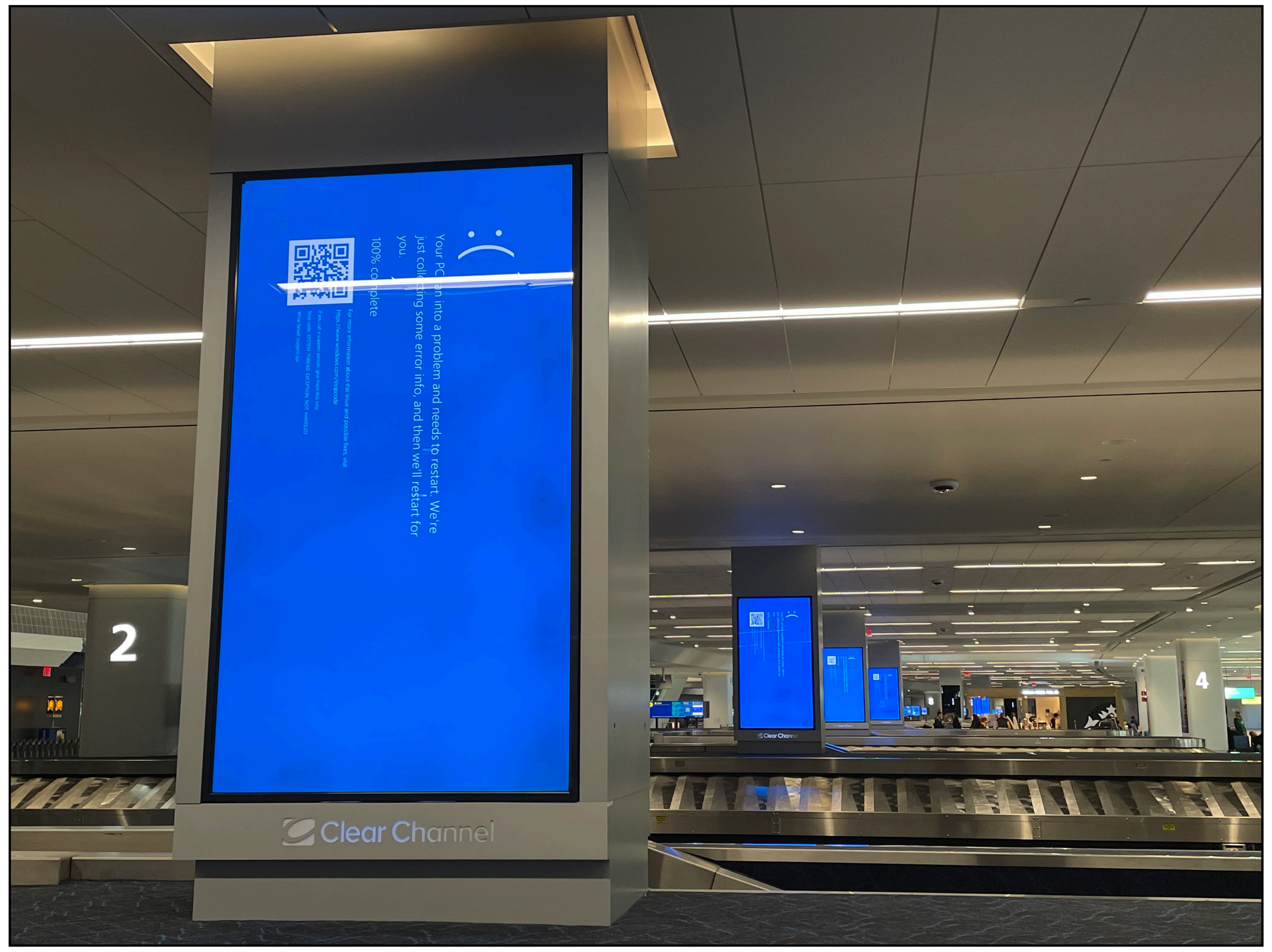
- February 2024: the White House issued a memorandum recommending memory safety and using formal methods

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

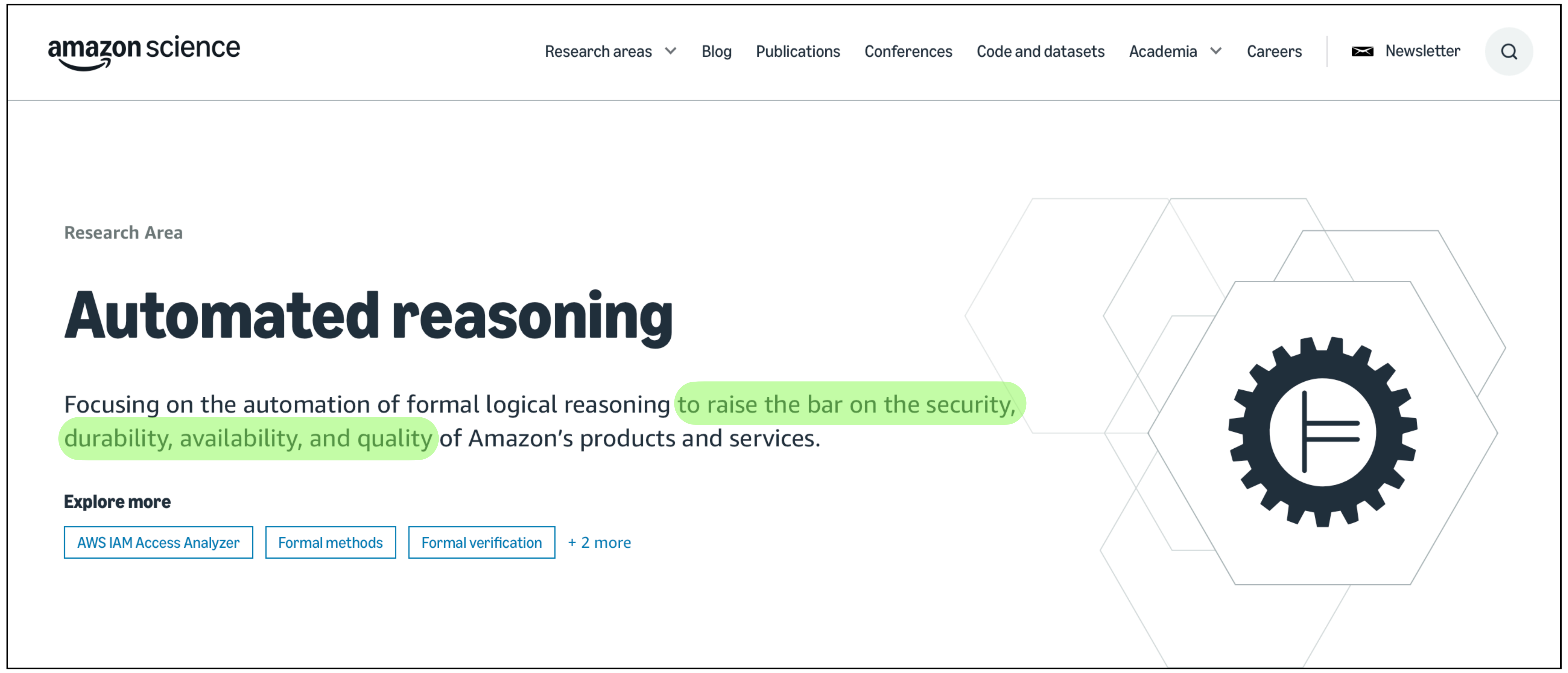
Blue screens of death at LGA airport from the CrowdStrike 2024 July outage



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 "→" that describes **individual steps of computation**
  - Captures enough details to rule out (the class of) bugs we are interested in

$$\textsf{if not true then } 2 \textsf{ else } 3 \rightarrow \textsf{if false then } 2 \textsf{ else } 3 \rightarrow 3$$

Operational Semantics

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

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

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

Operational Semantics

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

$$\textsf{if}\ \text{"a"}\ \textsf{then}\ 2\ \textsf{else}\ 3 \to ?$$

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

Getting stuck indicates an error

$$\textsf{if}\ \texttt{"a"}\ \textsf{then}\ 2\ \textsf{else}\ 3 \not\rightarrow$$

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)

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

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)

$$\mathsf{Safe}(e) := \forall e'.\ e \to^* e' \implies \mathsf{Val}(e') \lor \exists e''.\ e' \to e''$$

Zero or more steps

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

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

Safety and Partial Functional Correctness

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

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

Zero or more steps

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

- Example: $\mathsf{Safe}\Big(\big(\mathsf{rec}\ f\ x\ := \ \mathsf{if}\ x == 0\ \mathsf{then}\ 1\ \mathsf{else}\ x \times f\ (x-1)\big)\ \ 5\Big)$

- Counterexample: $\neg\mathsf{Safe}\left(\mathsf{if}\ \texttt{"a"}\ \mathsf{then}\ 2\ \mathsf{else}\ 3\right)$

Safety and Partial Functional Correctness

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

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

Zero or more steps

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

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

Safety and Partial Functional Correctness

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

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

Zero or more steps

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

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

Safety and Partial Functional Correctness

- **Partial Functional correctness**: safe & upon termination the postcondition ϕ holds

$$\text{Correct}_{\phi}(e) := \forall e' .\ e \to^{*} e' \implies (\text{Val}(e') \land \phi(e')) \lor \exists e'' .\ e' \to e''$$

The postcondition

Safety and Partial Functional Correctness

- **Partial Functional correctness**: safe & upon termination the postcondition ϕ holds

$$\mathsf{Correct}_{\phi}(e) := \forall e'.\ e \to^* e' \implies (\mathsf{Val}(e') \land \phi(e')) \lor \exists e''.\ e' \to e''$$

The postcondition

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

Safety and Partial Functional Correctness

- **Partial Functional correctness**: safe & upon termination the postcondition ϕ holds

$$\text{Correct}_\phi(e) := \forall e'.\ e \to^* e' \implies (\text{Val}(e') \land \phi(e')) \lor \exists e''.\ e' \to e''$$

The postcondition

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

- **Q**: What should ϕ be for $\text{Correct}_\phi\left(\text{rec } f\ x\ := \ \text{if } x == 0 \text{ then } 1 \text{ else } x \times f\ (x-1)\right)$?

Safety and Partial Functional Correctness

- **Partial Functional correctness**: safe & upon termination the postcondition ϕ holds

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

The postcondition

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

- **Q**: What should ϕ be for $\text{Correct}_{\phi}\left(\text{rec } f\ x\ := \ \text{if } x == 0 \text{ then } 1 \text{ else } x \times f\ (x-1)\right)$?

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

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 ℓ

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? $\textsf{if}\ !\ell < 0\ \textsf{then}\ 0\ \textsf{else}\ !\ell + 1$

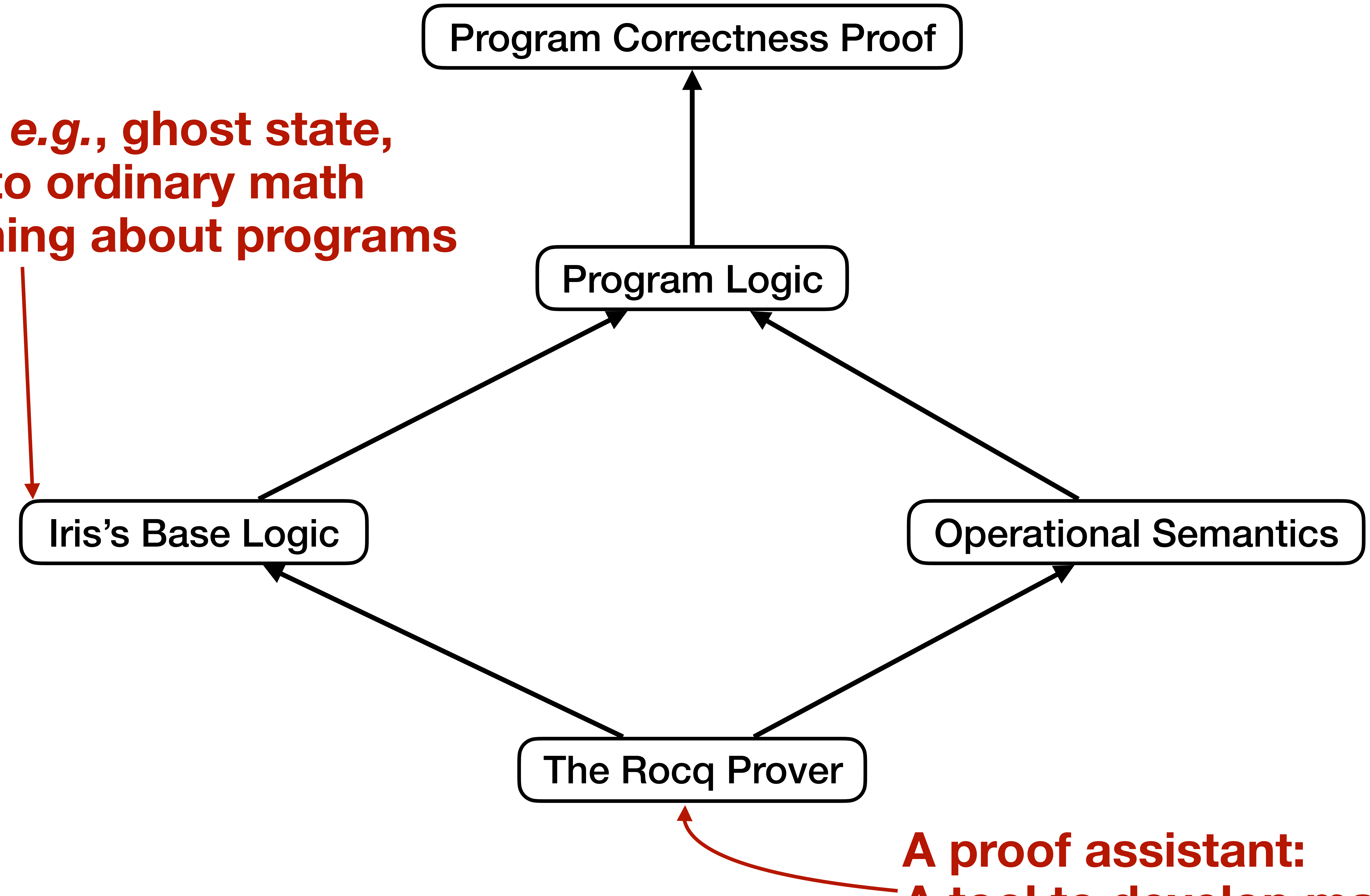
Loading from memory location ℓ

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

Program Logics: the Iris Framework

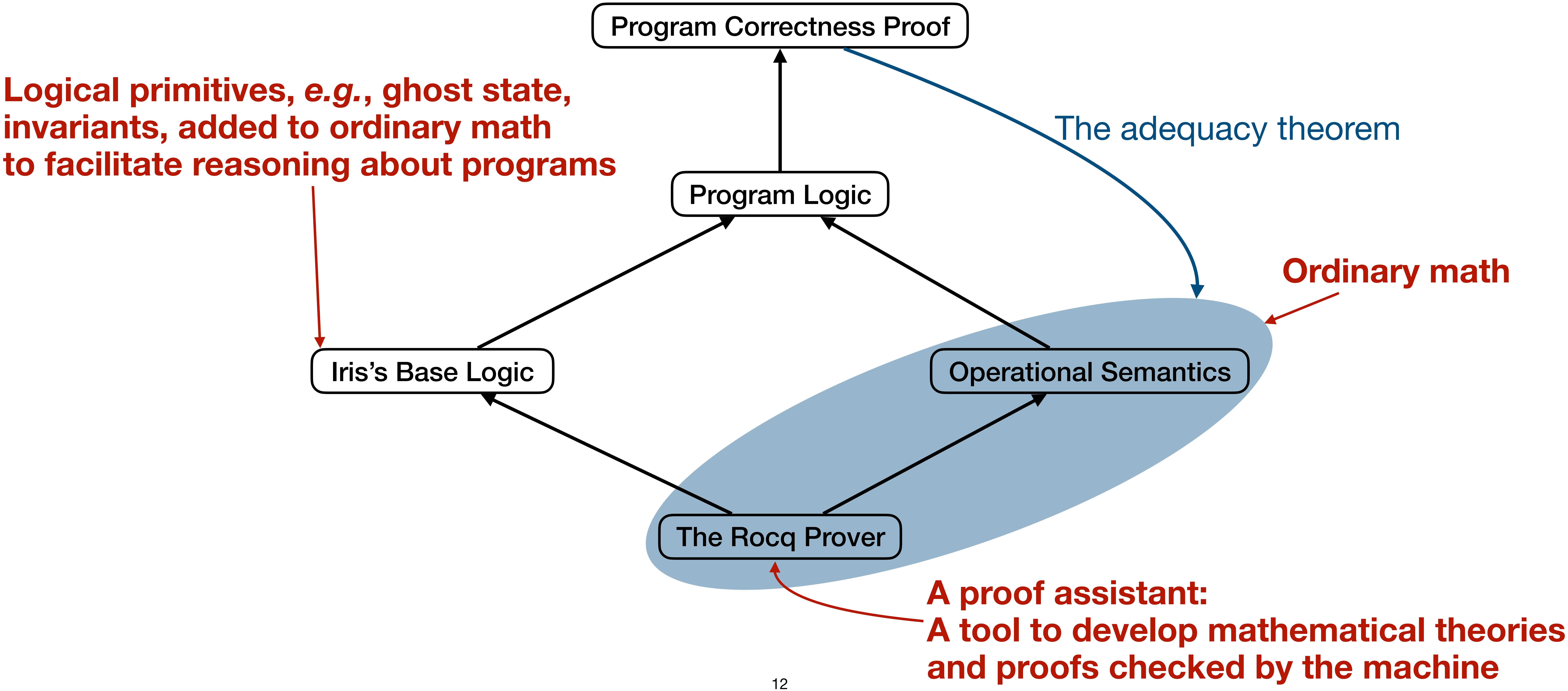
The Iris Framework

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

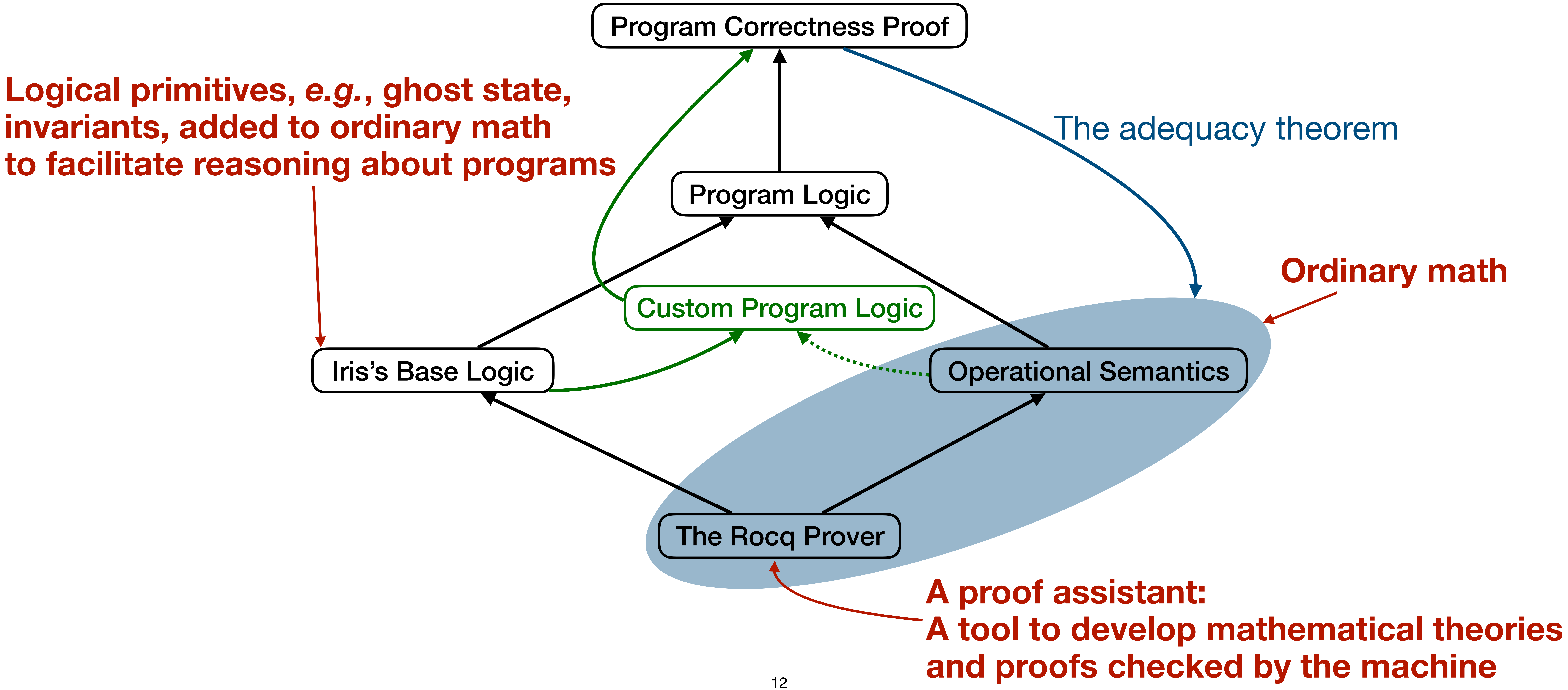


A proof assistant:
A tool to develop mathematical theories and proofs checked by the machine

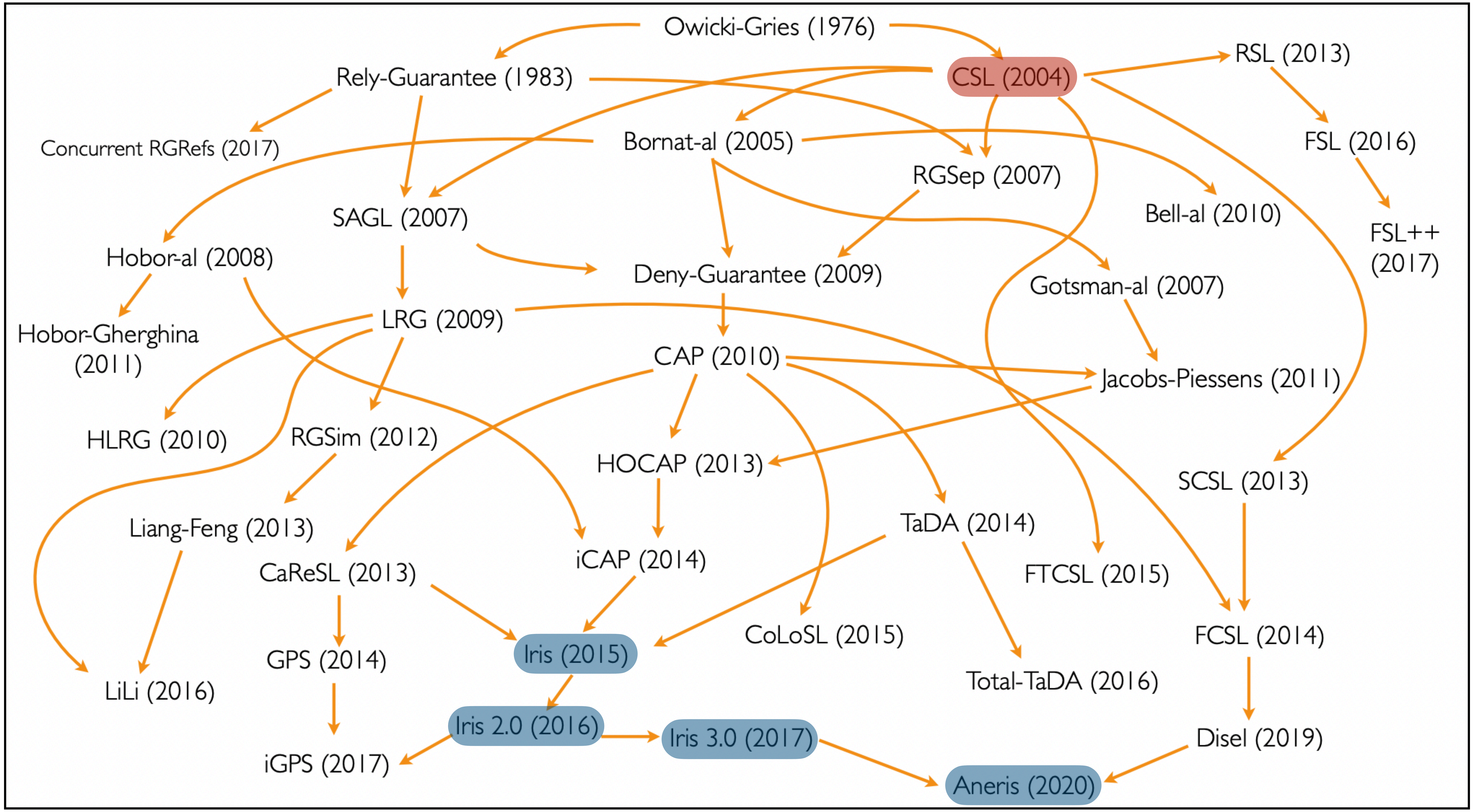
The Iris Framework



The Iris Framework



Program Logics for Concurrent and Distributed Programs



Taken from wikimedia.org, 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:

The program

The precondition → $\{P\}e\{x.\,Q\}$ ← **Binds the return value**

The postcondition

Examples:

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

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

Iris's Program Logic

A Higher-Order Separation Logic

A Hoare-style logic:

The program

The precondition

Binds the return value

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

The postcondition

Examples:

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

$$\{n \geq 0\}\left(\mathsf{rec}\ f\ x\ \ :=\ \ \mathsf{if}\ x == 0\ \mathsf{then}\ 1\ \mathsf{else}\ x \times f\ (x-1)\right)\ n\{x.\ x = n!\}$$

Iris's Program Logic

A Higher-Order Separation Logic

A Hoare-style logic:

The program

The precondition → Binds the return value

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

The postcondition

Examples:

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

$$\{n \geq 0\}\left(\mathsf{rec}\ f\ x\ := \ \mathsf{if}\ x == 0\ \mathsf{then}\ 1\ \mathsf{else}\ x \times f\ (x - 1)\right)\ n\{x.\ x = n!\}$$

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

The Adequacy Theorem

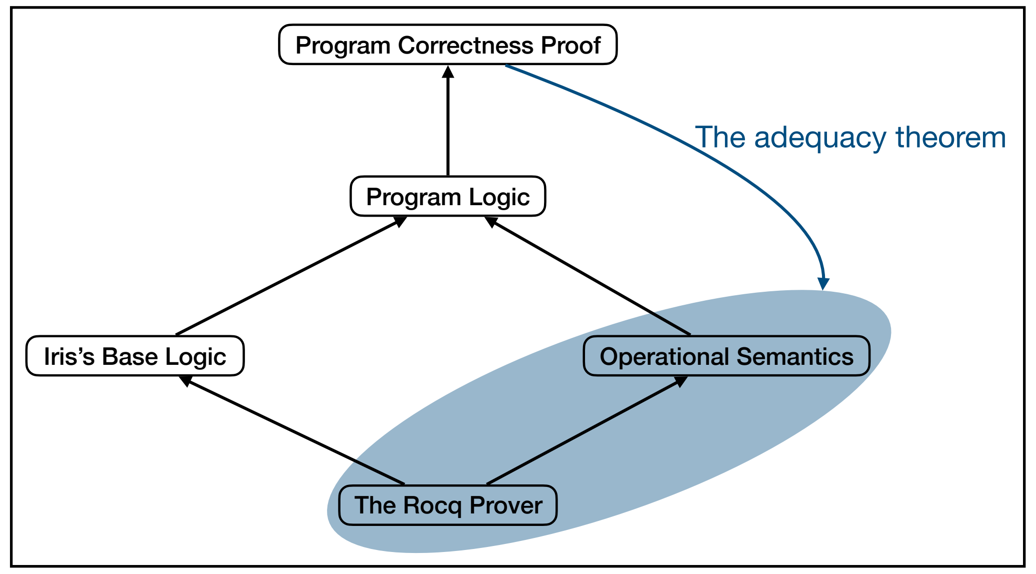
$$\{\mathsf{True}\}2 + 3\{x\,.\,isOdd(x)\}$$

Theorem (Adequacy)

If we prove

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

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



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

Separation Logic

Separating Conjunction:

the separating conjunction

$$P \star Q$$

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

Separation Logic

Separating Conjunction:

the separating conjunction

$$P \star Q$$

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

Example: exclusive ownership of a memory location

the points-to proposition

$$\ell \mapsto v$$

Separation Logic

Separating Conjunction:

the separating conjunction

$$P \star Q$$

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

Example: exclusive ownership of a memory location

the points-to proposition

$$\ell \mapsto v$$

Points-to-Exclusive

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

Hoare-Alloc

$$\{\mathsf{True}\}\,\mathsf{ref}\ v\{\ell\,.\,\ell \mapsto v\}$$

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

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

Concurrent Separation Logic

Disjoint Concurrency

Threads working on disjoint resources can be safely run in parallel

Hoare-Par

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

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

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

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

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

Concurrent Separation Logic

Shared Memory Concurrency

Consider the following program

$$\begin{array}{l}\textsf{let}\ c = \textsf{ref}\ 0\ \textsf{in} \\ \quad (\textsf{faa}\ c\ 1 \parallel \textsf{faa}\ c\ 2); \\ \quad !c\end{array}$$

Concurrent Separation Logic

Shared Memory Concurrency

Consider the following program

$$\begin{array}{l} \textsf{let}\ c = \textsf{ref}\ 0\ \textsf{in} \\ \quad (\textsf{faa}\ c\ 1 \parallel \textsf{faa}\ c\ 2); \\ \quad !c \end{array}$$

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

Concurrent Separation Logic

Shared Memory Concurrency

Consider the following program

$$\{\mathsf{True}\}$$
$$\mathsf{let}\ c = \mathsf{ref}\ 0\ \mathsf{in}$$
$$(\mathsf{faa}\ c\ 1 \parallel \mathsf{faa}\ c\ 2);$$
$$!c$$
$$\{x\,.\,x \geq 0\}$$

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

$$\{\mathsf{True}\}$$
$$\begin{array}{l} \mathsf{let}\ c = \mathsf{ref}\ 0\ \mathsf{in} \\ \quad (\mathsf{faa}\ c\ 1 \parallel \mathsf{faa}\ c\ 2); \\ \quad !c \end{array}$$
$$\{x \,.\, x \geq 0\}$$

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”

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

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

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

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

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

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

$$!c$$

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

Concurrent Separation Logic

Shared Memory Concurrency: A Stronger Postcondition

What if we wish to prove something stronger?

$$\{\mathsf{True}\}$$
$$\begin{array}{l} \mathsf{let}\ c = \mathsf{ref}\ 0\ \mathsf{in} \\ \quad (\mathsf{faa}\ c\ 1 \parallel \mathsf{faa}\ c\ 2); \\ \quad !c \end{array}$$
$$\{x \,.\, x = 3\}$$

Q: What invariant should we take now?

Concurrent Separation Logic

Shared Memory Concurrency: A Stronger Postcondition

What if we wish to prove something stronger?

$$\{\mathsf{True}\}$$
$$\mathsf{let}\ c = \mathsf{ref}\ 0\ \mathsf{in}$$
$$(\mathsf{faa}\ c\ 1 \parallel \mathsf{faa}\ c\ 2);$$
$$!c$$
$$\{x.\, x = 3\}$$

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:

$$\mathsf{Left}^{\gamma}(n) \qquad \mathsf{Sum}^{\gamma}(k) \qquad \mathsf{Right}^{\gamma}(m)$$

The left thread's contribution to the value of c is n

The sum of contributions is k

The right thread's contribution to the value of c is m

Concurrent Separation Logic

Ghost State

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

$$\mathsf{Left}^{\gamma}(n) \qquad \mathsf{Sum}^{\gamma}(k) \qquad \mathsf{Right}^{\gamma}(m)$$

The left thread's contribution to the value of c is n

The sum of contributions is k

The right thread's contribution to the value of c is m

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

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

Concurrent Separation Logic

Ghost State

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

$$\mathsf{Left}^{\gamma}(n) \qquad \mathsf{Sum}^{\gamma}(k) \qquad \mathsf{Right}^{\gamma}(m)$$

The left thread's contribution to the value of c is n

The sum of contributions is k

The right thread's contribution to the value of c is m

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

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

And, after the two threads are run:

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

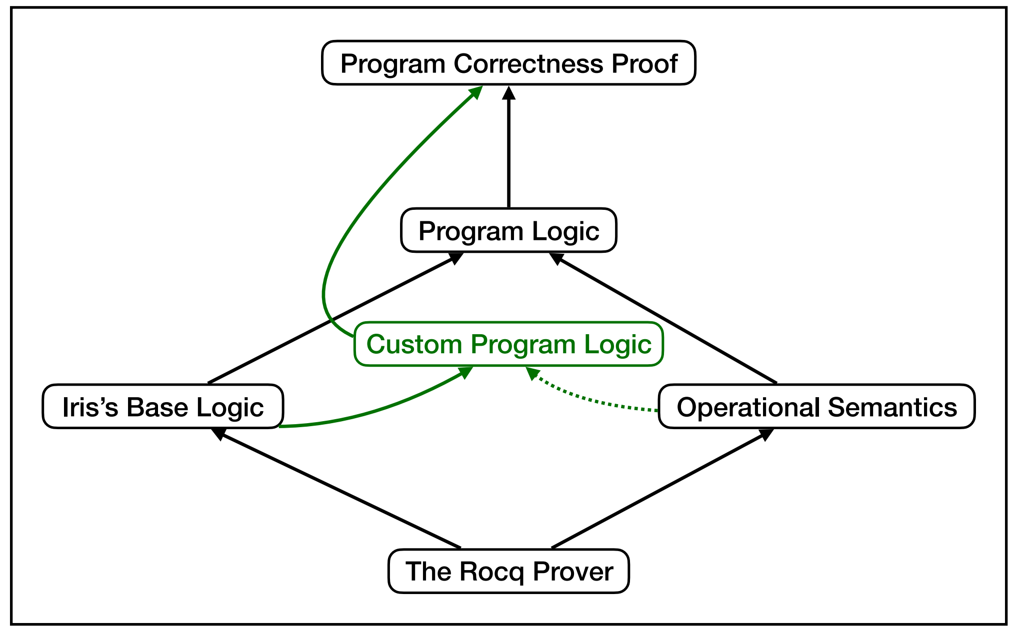
Verification of Distributed Systems

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"

The address → $a \Mapsto \Phi$ ← **The socket protocol, a predicate on messages**



Aneris is one such custom program logic

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

Modular Verification of Distributed Systems

Applications of the Aneris Program Logic

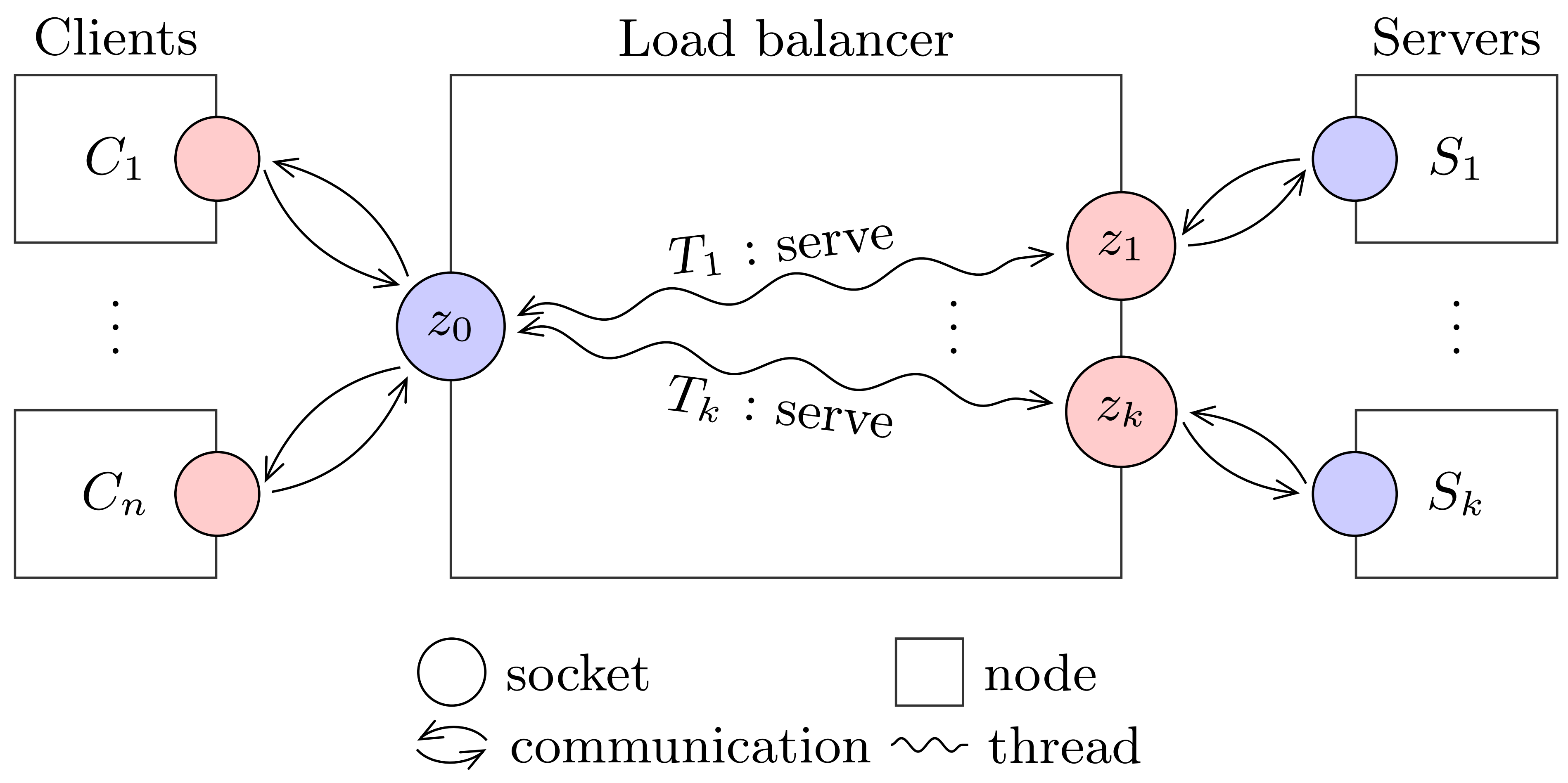
We have used Aneris to verify of the following distributed systems and their clients

- **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

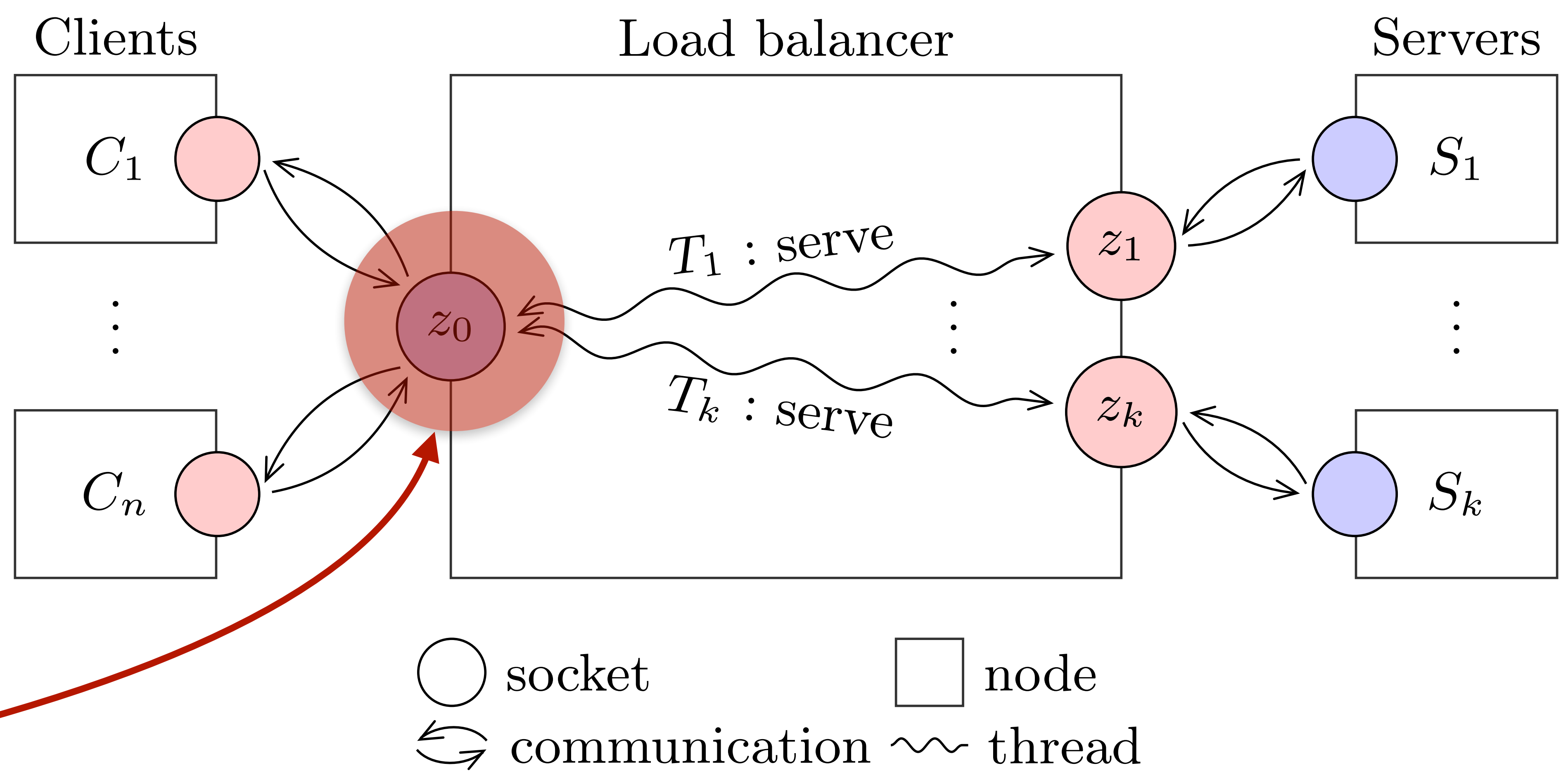
- A verified load balancer



Verification of Distributed Systems

The Aneris Program Logic

- A verified load balancer



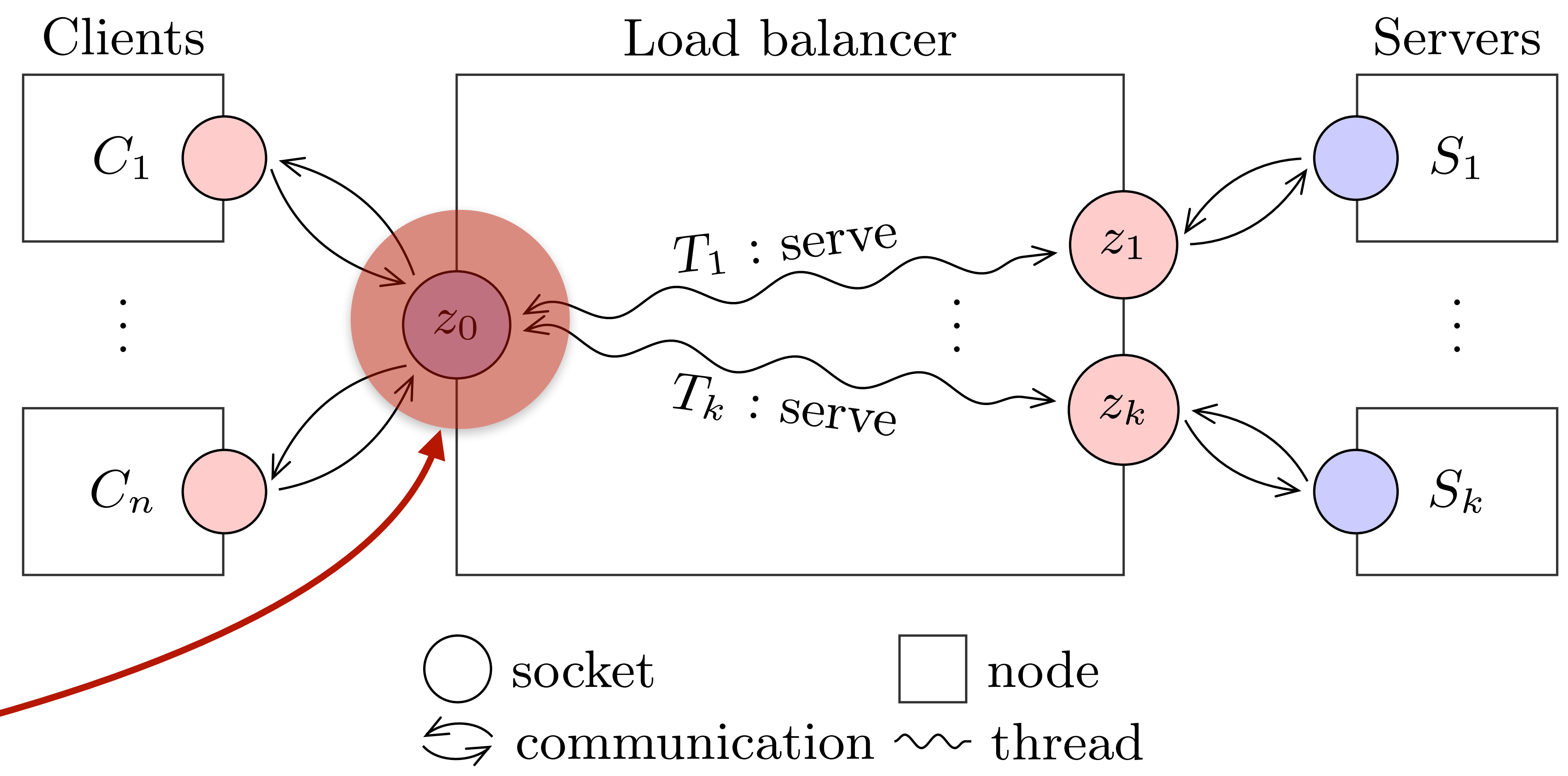
What should the socket protocol be?

- **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 Ψ accepts server's response

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

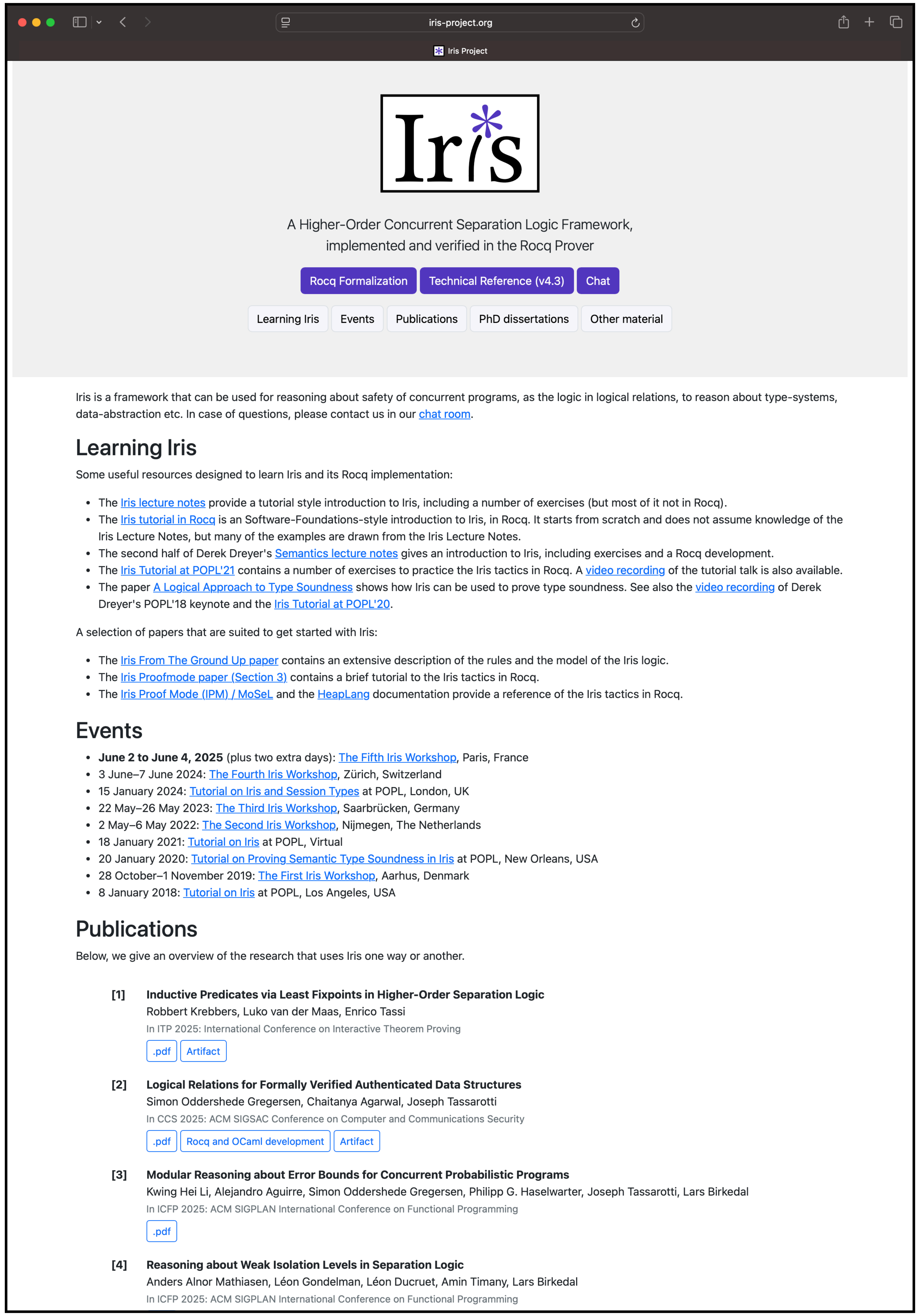
Higher-order: quantifies over protocols to define a protocol

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



Thanks!