# Higher-Order Concurrent Separation Logic

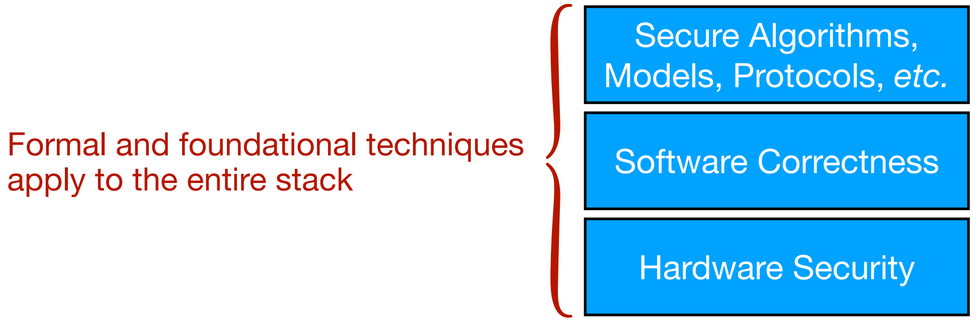
## The PLS Summer School

Amin Timany
August 12, 2026

# Why Verify Programs?

**Software correctness is essential for security**

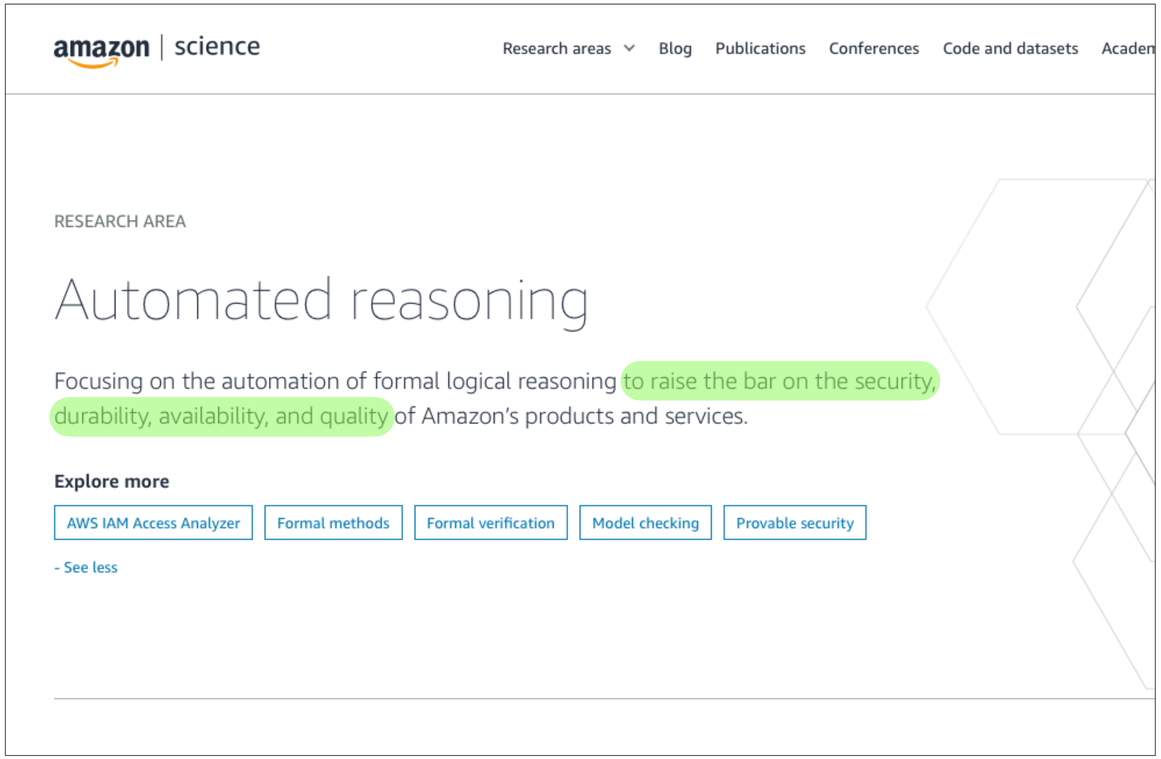
- Bugs, compromising security can occur in implementations, even if the high-level models and protocols are correct.
  - Example: the infamous Hearbleed bug
    - A memory safety bug
  - The entire stack should be secure



# Why Study 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



# An Overview of Formal Program Verification

# How Do We Prove a Program Correct?

**This program should compute n!**

```
// assume n > 0, i = 1, f = 1
while(i <= n){
    f = f * i;
    i = i + 1;
}
// f = n!
```

# How Do We Prove a Program Correct?

**This program should compute n!**

**Key Insight:**

**For each statement C,
if the condition above C holds,
then after running C the condition below it holds**

```
// assume n > 0, i = 1, f = 1
```

$$n > 0 \wedge f = (i-1)! \wedge i \leq n+1$$

```
while(i <= n){
```

$$n > 0 \wedge f = (i-1)! \wedge i \leq n$$

```
    f = f * i;
```

$$n > 0 \wedge f = i! \wedge i \leq n$$

```
    i = i + 1;
```

$$n > 0 \wedge f = (i-1)! \wedge i \leq n+1$$

```
}
```

$$n > 0 \wedge f = (i-1)! \wedge i \leq n+1 \wedge i > n$$

```
// f = n!
```

# How Do We Prove a Program Correct?

## This program should compute n!

**Key Insight:**

**For each statement C,
if the condition above C holds,
then after running C the condition below it holds**

```
// assume n > 0, i = 1, f = 1
```

Generalized

$$n > 0 \wedge f = (i-1)! \wedge i \leq n+1$$

```
while(i <= n){
```

$$n > 0 \wedge f = (i-1)! \wedge i \leq n$$

```
    f = f * i;
```

$$n > 0 \wedge f = i! \wedge i \leq n$$

```
    i = i + 1;
```

$$n > 0 \wedge f = (i-1)! \wedge i \leq n+1$$

```
}
```

$$n > 0 \wedge f = (i-1)! \wedge i \leq n+1 \wedge i > n$$

```
// f = n!
```

# How Do We Prove a Program Correct?

## This program should compute n!

**Key Insight:**

**For each statement C,
if the condition above C holds,
then after running C the condition below it holds**

```
// assume n > 0, i = 1, f = 1
```

$$n > 0 \wedge f = (i-1)! \wedge i \leq n+1$$

```
while(i <= n){
```

$$n > 0 \wedge f = (i-1)! \wedge i \leq n$$

```
    f = f * i;
```

$$n > 0 \wedge f = i! \wedge i \leq n$$

```
    i = i + 1;
```

$$n > 0 \wedge f = (i-1)! \wedge i \leq n+1$$

***Loop Invariant***

```
}
```

$$n > 0 \wedge f = (i-1)! \wedge i \leq n+1 \wedge i > n$$

```
// f = n!
```

# How Do We Prove a Program Correct?

## This program should compute n!

```
// assume n > 0, i = 1, f = 1
```

$$n > 0 \wedge f = (i-1)! \wedge i \leq n+1$$

```
while(i <= n){
```

$$n > 0 \wedge f = (i-1)! \wedge i \leq n$$

```
    f = f * i;
```

$$n > 0 \wedge f = i! \wedge i \leq n$$

```
    i = i + 1;
```

$$n > 0 \wedge f = (i-1)! \wedge i \leq n+1$$

```
}
```

$$n > 0 \wedge f = (i-1)! \wedge i \leq n+1 \wedge i > n$$

```
// f = n!
```

Same as before the loop + the condition holds

**Key Insight:**

**For each statement C,
if the condition above C holds,
then after running C the condition below it holds**

# How Do We Prove a Program Correct?

**This program should compute n!**

**Key Insight:**

**For each statement C,**
**if the condition above C holds,**
**then after running C the condition below it holds**

```
// assume n > 0, i = 1, f = 1
```

$$n > 0 \wedge f = (i-1)! \wedge i \leq n+1$$

```
while(i <= n){
```

$$n > 0 \wedge f = (i-1)! \wedge i \leq n$$

```
    f = f * i;
```

$$n > 0 \wedge f = i! \wedge i \leq n$$

```
    i = i + 1;
```

$$n > 0 \wedge f = (i-1)! \wedge i \leq n+1$$

```
}
```

$$n > 0 \wedge f = (i-1)! \wedge i \leq n+1 \wedge i > n$$

```
// f = n!
```

**Same as before the loop + the condition doesn't hold**

# How Do We Prove a Program Correct?

**This program should compute n!**

**Key Insight:**

**For each statement C,**
**if the condition above C holds,**
**then after running C the condition below it holds**

```
// assume n > 0, i = 1, f = 1
```

$$n > 0 \wedge f = (i-1)! \wedge i \leq n+1$$

```
while(i <= n){
```

$$n > 0 \wedge f = (i-1)! \wedge i \leq n$$

```
    f = f * i;
```

$$n > 0 \wedge f = i! \wedge i \leq n$$

```
    i = i + 1;
```

$$n > 0 \wedge f = (i-1)! \wedge i \leq n+1$$

```
}
```

$$n > 0 \wedge f = (i-1)! \wedge i \leq n+1 \wedge i > n$$

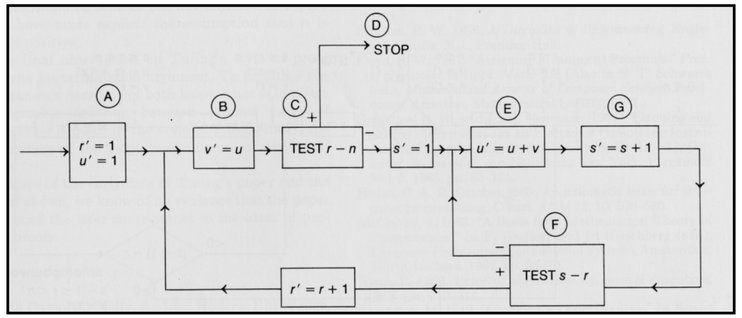
**Implies**

```
// f = n!
```

# Factorial: The First Program Verified

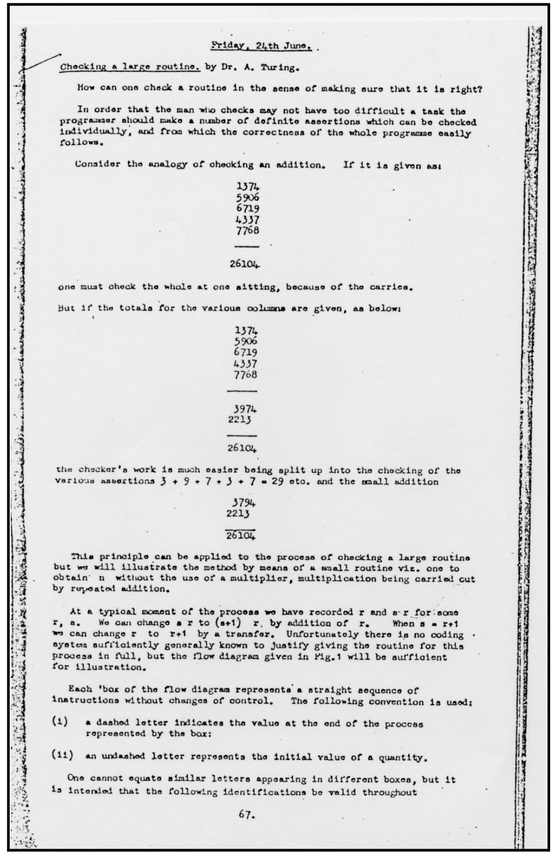
## [Turing 1949]: Checking a Large Routine

- Verifies a factorial program with two nested loops
- Uses a “flow diagram” to write the program




Unfortunately there is no coding system sufficiently generally known to justify giving the routine for this process in full, but the flow diagram given in Fig. 1 will be sufficient for illustration.

- Turing’s work goes unnoticed for decades



Friday, 24th June.

Checking a large routine. by Dr. A. Turing.

How can one check a routine in the sense of making sure that it is right?

In order that the man who checks may not have too difficult a task the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole programme easily follows.

Consider the analogy of checking an addition. If it is given as:

1374
5906
6719
4337
7768
26104

one must check the whole at one sitting, because of the carries.

But if the totals for the various columns are given, as below:

1374
5906
6719
4337
7768
3974
2213
26104

the checker's work is much easier being split up into the checking of the various assertions 3 + 9 + 7 + 3 + 7 = 29 etc. and the small addition

3794
2213
26104

This principle can be applied to the process of checking a large routine but we will illustrate the method by means of a small routine viz. one to obtain n without the use of a multiplier, multiplication being carried out by repeated addition.

At a typical moment of the process we have recorded r and s = r for some r, s. We can change s r to (s+1) r, by addition of r. When s = r+1 we can change r to r+1 by a transfer. Unfortunately there is no coding system sufficiently generally known to justify giving the routine for this process in full, but the flow diagram given in Fig.1 will be sufficient for illustration.

Each 'box of the flow diagram represents a straight sequence of instructions without changes of control. The following convention is used:

(i) a dashed letter indicates the value at the end of the process represented by the box:

(ii) an undashed letter represents the initial value of a quantity.

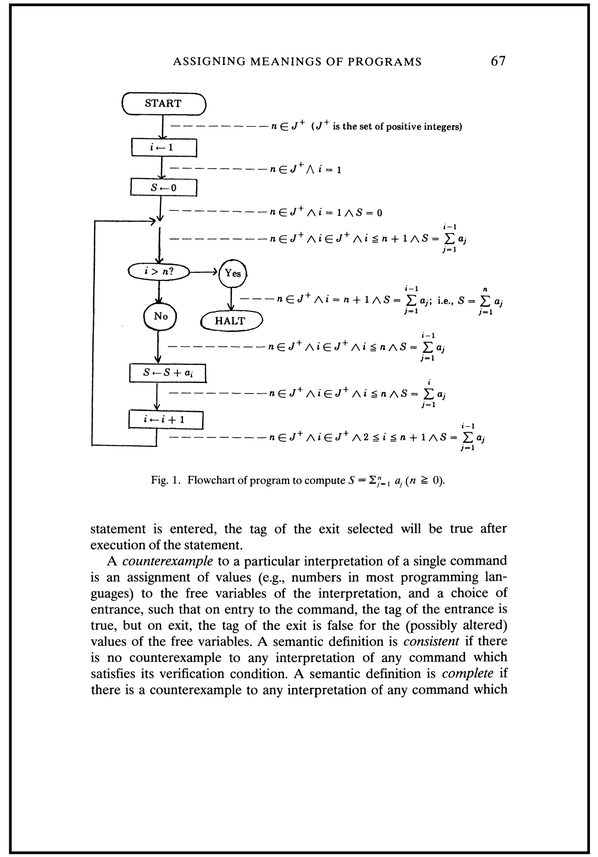
One cannot equate similar letters appearing in different boxes, but it is intended that the following identifications be valid throughout

67.

# Floyd Independently Rediscovers Turing's Ideas

## [Floyd 1967]: Assigning Meaning to Programs

- Introduces verification conditions: $V_c(P, Q)$
- Describes how to massage verification conditions so that they match (can be laid on a flow chart)



ASSIGNING MEANINGS OF PROGRAMS 67



Fig. 1. Flowchart of program to compute $S = \Sigma_{j=1}^{n} a_j$ ($n \geq 0$).

statement is entered, the tag of the exit selected will be true after execution of the statement.

A *counterexample* to a particular interpretation of a single command is an assignment of values (e.g., numbers in most programming languages) to the free variables of the interpretation, and a choice of entrance, such that on entry to the command, the tag of the entrance is true, but on exit, the tag of the exit is false for the (possibly altered) values of the free variables. A semantic definition is *consistent* if there is no counterexample to any interpretation of any command which satisfies its verification condition. A semantic definition is *complete* if there is a counterexample to any interpretation of any command which

# Hoare Builds on Floyd's work

## [Hoare 1969]: An Axiomatic Basis for Computer Programming

- Abandons flow charts
- An axiomatic system based on "Hoare triples": $\{P\}e\{Q\}$


$$\frac{\{P\}e_1\{Q\} \qquad \{Q\}e_2\{R\}}{\{P\}e_1; e_2\{R\}}$$

$$\frac{\{I \wedge B\}e\{I\}}{\{I\}\texttt{While}(B)\texttt{do } e\{I \wedge \neg B\}}$$

# Hoare Builds on Floyd's work

**[Hoare 1969]: An Axiomatic Basis for Computer Programming**

- Abandons flow charts
- An axiomatic system based on "Hoare triples": $\{P\}e\{Q\}$


$$\frac{\{P\}e_1\{Q\} \qquad \{Q\}e_2\{R\}}{\{P\}e_1; e_2\{R\}} \qquad\qquad \frac{\{I \wedge B\}e\{I\}}{\{I\}\texttt{While}(B)\texttt{do } e\{I \wedge \neg B\}}$$

- Hoare emphasized modularity
  - Decompose the problem into smaller, more manageable parts
  - Proof reuse: verify a function once, use its specs at different call sites

# Hoare Builds on Floyd's work

**[Hoare 1969]: An Axiomatic Basis for Computer Programming**

- Abandons flow charts
- An axiomatic system based on "Hoare triples": $\{P\}e\{Q\}$


$$\frac{\{P\}e_1\{Q\} \qquad \{Q\}e_2\{R\}}{\{P\}e_1; e_2\{R\}} \qquad\qquad \frac{\{I \wedge B\}e\{I\}}{\{I\}\texttt{While}(B)\texttt{do } e\{I \wedge \neg B\}}$$

- Hoare emphasized modularity
  - Decompose the problem into smaller, more manageable parts
  - Proof reuse: verify a function once, use its specs at different call sites

**Essential: a strong specification language (logic) to express modules' contracts**

# Separation Logic: Reasoning About Aliasing

**[Reynolds 2002]: Separation Logic: A Logic for Shared Mutable Data Structures**
**[O'Hearn 2004]: Resources, Concurrency and Local Reasoning**

- Our earlier argument was not very rigorous
- We are implicitly relying on non-aliasing
  - Memory storing $n$, $f$, and $i$ are **disjoint**

```
// assume n > 0, i = 1, f = 1
```
$$n > 0 \wedge f = (i-1)! \wedge i \leq n+1$$
```
while(i <= n){
```
$$n > 0 \wedge f = (i-1)! \wedge i \leq n$$
```
    f = f * i;
```
$$n > 0 \wedge f = i! \wedge i \leq n$$
```
    i = i + 1;
```
$$n > 0 \wedge f = (i-1)! \wedge i \leq n+1$$
```
}
```
$$n > 0 \wedge f = (i-1)! \wedge i \leq n+1 \wedge i > n$$
```
// f = n!
```

# Separation Logic: Reasoning About Aliasing

**[Reynolds 2002]: Separation Logic: A Logic for Shared Mutable Data Structures**
**[O'Hearn 2004]: Resources, Concurrency and Local Reasoning**

```
// assume n > 0, i = 1, f = 1
```

$$n > 0 \wedge f = (i-1)! \wedge i \leq n+1$$

```
while(i <= n){
```

$$n > 0 \wedge f = (i-1)! \wedge i \leq n$$

```
    f = f * i;
```

$$n > 0 \wedge f = i! \wedge i \leq n$$

```
    i = i + 1;
```

$$n > 0 \wedge f = (i-1)! \wedge i \leq n+1$$

```
}
```

$$n > 0 \wedge f = (i-1)! \wedge i \leq n+1 \wedge i > n$$

```
// f = n!
```

# Separation Logic: Reasoning About Aliasing

**[Reynolds 2002]: Separation Logic: A Logic for Shared Mutable Data Structures**
**[O'Hearn 2004]: Resources, Concurrency and Local Reasoning**

**Key Points:**

**Distinguish between address and value: $\hookrightarrow$**

**Separating conjunction: $\star$**

```
// assume n > 0, i = 1, f = 1
```

$$n \hookrightarrow \mathbb{K} \star \mathbb{K} > 0 \star f \hookrightarrow (\mathbb{L}-1)! \star i \hookrightarrow \mathbb{L} \star \mathbb{L} \leq \mathbb{K}+1$$

```
while(i <= n){
```

$$n \hookrightarrow \mathbb{K} \star \mathbb{K} > 0 \star f \hookrightarrow (\mathbb{L}-1)! \star i \hookrightarrow \mathbb{L} \star \mathbb{L} \leq \mathbb{K}$$

```
    f = f * i;
```

$$n \hookrightarrow \mathbb{K} \star \mathbb{K} > 0 \star f \hookrightarrow \mathbb{L}! \star i \hookrightarrow \mathbb{L} \star \mathbb{L} \leq \mathbb{K}$$

```
    i = i + 1;
```

$$n \hookrightarrow \mathbb{K} \star \mathbb{K} > 0 \star f \hookrightarrow \mathbb{L}! \star i \hookrightarrow \mathbb{L}+1 \star \mathbb{L} \leq \mathbb{K}$$

```
}
```

$$n \hookrightarrow \mathbb{K} \star \mathbb{K} > 0 \star f \hookrightarrow (\mathbb{L}-1)! \star i \hookrightarrow \mathbb{L} \star \mathbb{L} \leq \mathbb{K}+1 \star \mathbb{L} > \mathbb{K}$$

```
// f = n!
```

# Separation Logic: Reasoning About Aliasing

**A strong specification language (logic) to express modules' contracts**

- Consider a function `swap` that swaps the values stored in its arguments

**Swap's specs:**

$$\{Stores(l_1, v_1) \wedge Stores(l_2, v_2)\}$$
$$swap(l_1, l_2)$$
$$\{Stores(l_1, v_2) \wedge Stores(l_2, v_1)\}$$

# Separation Logic: Reasoning About Aliasing

**A strong specification language (logic) to express modules' contracts**

- Consider a function `swap` that swaps the values stored in its arguments

**Swap's specs:**

$$\{Stores(l_1, v_1) \wedge Stores(l_2, v_2)\}$$
$$swap(l_1, l_2)$$
$$\{Stores(l_1, v_2) \wedge Stores(l_2, v_1)\}$$

**A client of swap:**

$Stores(l,10) \wedge Stores(k,2) \wedge Stores(p,10)$

```
swap(l, k);
```

$Stores(l,2) \wedge Stores(k,10) \wedge Stores(p,??)$

# Separation Logic: Reasoning About Aliasing

**A strong specification language (logic) to express modules' contracts**

- Consider a function `swap` that swaps the values stored in its arguments

**Swap's specs:**

$$\{Stores(l_1, v_1) \land Stores(l_2, v_2)\}$$
$$swap(l_1, l_2)$$
$$\{Stores(l_1, v_2) \land Stores(l_2, v_1)\}$$

**Swap's sep. log. specs:**

$$\{l_1 \hookrightarrow v_1 \star l_2 \hookrightarrow v_2\}$$
$$swap(l_1, l_2)$$
$$\{l_1 \hookrightarrow v_2 \star l_2 \hookrightarrow v_1\}$$

**A client of swap:**

$Stores(l,10) \land Stores(k,2) \land Stores(p,10)$

```
swap(l, k);
```

$Stores(l,2) \land Stores(k,10) \land Stores(p,??)$

# Separation Logic: Reasoning About Aliasing

**A strong specification language (logic) to express modules' contracts**

- Consider a function `swap` that swaps the values stored in its arguments

**Swap's specs:**

$$\{Stores(l_1, v_1) \land Stores(l_2, v_2)\}$$
$$swap(l_1, l_2)$$
$$\{Stores(l_1, v_2) \land Stores(l_2, v_1)\}$$

**Swap's sep. log. specs:**

$$\{l_1 \hookrightarrow v_1 \star l_2 \hookrightarrow v_2\}$$
$$swap(l_1, l_2)$$
$$\{l_1 \hookrightarrow v_2 \star l_2 \hookrightarrow v_1\}$$

**A client of swap:**

$Stores(l,10) \land Stores(k,2) \land Stores(p,10)$

```
swap(l, k);
```

$Stores(l,2) \land Stores(k,10) \land Stores(p,??)$

**A client of swap in sep. log.:**

$l \hookrightarrow 10 \star k \hookrightarrow 2 \star p \hookrightarrow 10$

```
swap(l, k);
```

$l \hookrightarrow 2 \star k \hookrightarrow 10 \star p \hookrightarrow 10$

# Separation Logic: Reasoning About Aliasing

**A strong specification language (logic) to express modules' contracts**

**Swap's sep. log. specs:**

$$\{l_1 \hookrightarrow v_1 \star l_2 \hookrightarrow v_2\}$$
$$swap(l_1, l_2)$$
$$\{l_1 \hookrightarrow v_2 \star l_2 \hookrightarrow v_1\}$$

**A client of swap in sep. log.:**

$l \hookrightarrow 10 \star k \hookrightarrow 2 \star p \mapsto 10$

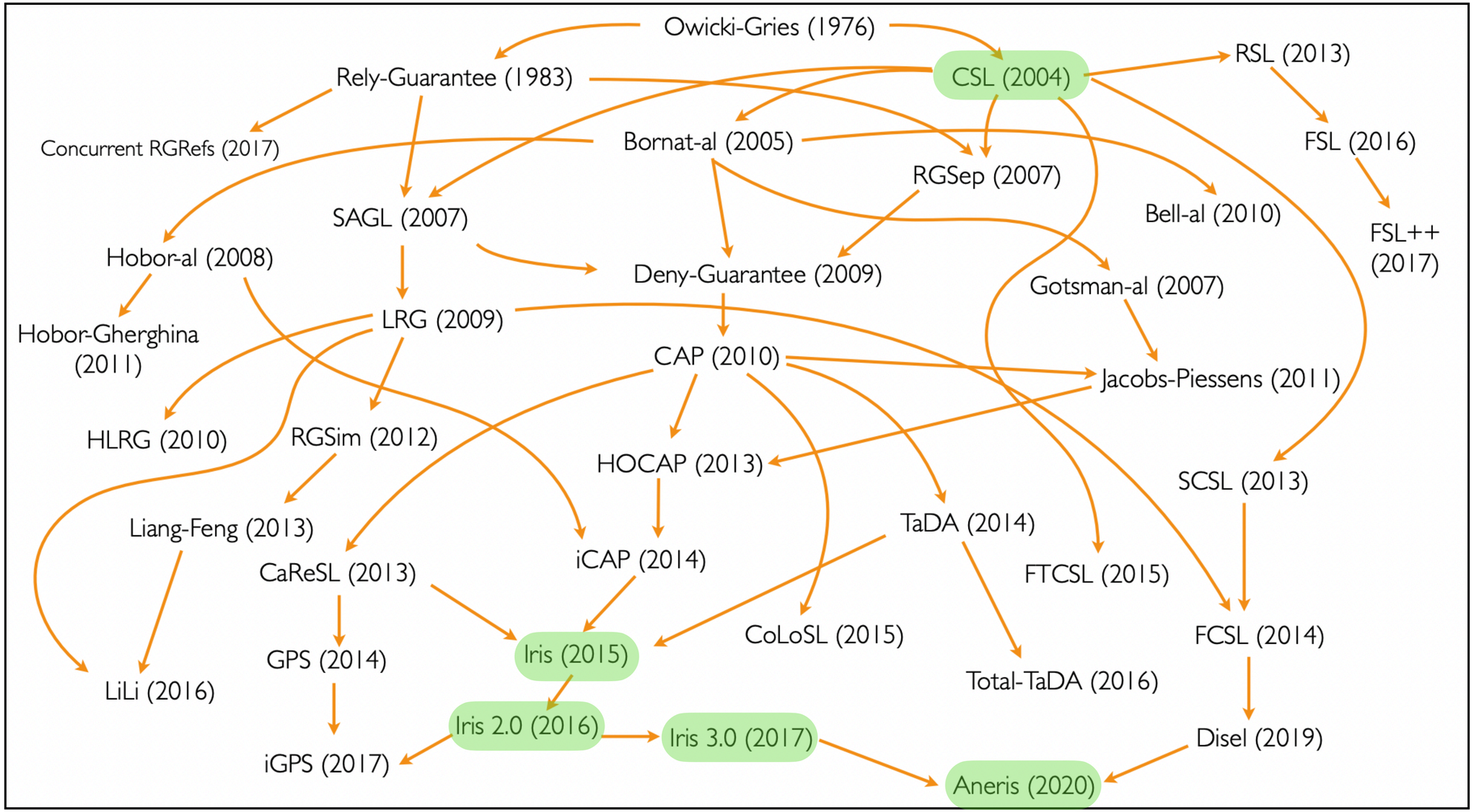
```
swap(l, k);
```

$l \hookrightarrow 2 \star k \hookrightarrow 10 \star p \hookrightarrow 10$

- Separation logic specs
  - The separating conjunction, ★, implicitly captures non-aliasing
  - The precondition is an over-approximation of **program's footprint**, that is, any memory that the program may read or write.[1]

1. This is besides the memory shared by multiple threads; we will later see how concurrent separation logic deals with that.

# Verification of 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.**

# The Foundations

## Operational Semantics, Safety, and Functional Correctness

# Operational Semantics

- We take a simple (OCaml-like) functional imperative language, also known as HeapLang, or $\lambda_{\text{ref,conc}}$
- 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, also known as HeapLang, or $\lambda_{\text{ref,conc}}$
- 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, also known as HeapLang, or $\lambda_{\text{ref,conc}}$
- Define a relation "$\rightarrow$" that describes **individual steps of computation**
  - Captures enough details to rule out (the class of) bugs we are interested in

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

# Operational Semantics

- We take a simple (OCaml-like) functional imperative language, also known as HeapLang, or $\lambda_{\text{ref,conc}}$
- Define a relation "$\rightarrow$" 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, also known as HeapLang, or $\lambda_{\text{ref,conc}}$
- Define a relation "$\rightarrow$" that describes **individual steps of computation**
  - Captures enough details to rule out (the class of) bugs we are interested in

$$\texttt{if "a" then } 2 \texttt{ else } 3 \rightarrow \ ?$$

# Operational Semantics

- We take a simple (OCaml-like) functional imperative language, also known as HeapLang, or $\lambda_{\text{ref,conc}}$
- 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**

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

**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)

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

**Zero or more steps**

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

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

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

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

# Safety and Partial Functional Correctness

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

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

**Zero or more steps**

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

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

# Safety and Partial Functional Correctness

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

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

The postcondition

# Safety and Partial Functional Correctness

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

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

The postcondition

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

# Safety and Partial Functional Correctness

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

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

The postcondition

- Example: $\text{Correct}_{isOdd}(2+3)$
- **Q**: What should $\phi$ 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 $\phi$ holds

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

The postcondition

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

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

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!
  - 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

**Answer: Hoare Logic**

# 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

**Answer: Hoare Logic**

- How do we even specify correctness of memory accesses?
  - Is the following program safe? $\text{if } !\ell < 0 \text{ then } 0 \text{ else } !\ell + 1$

**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

**Answer: Hoare Logic**

- How do we even specify correctness of memory accesses?
  - Is the following program safe? $\text{if } !\ell < 0 \text{ then } 0 \text{ else } !\ell + 1$

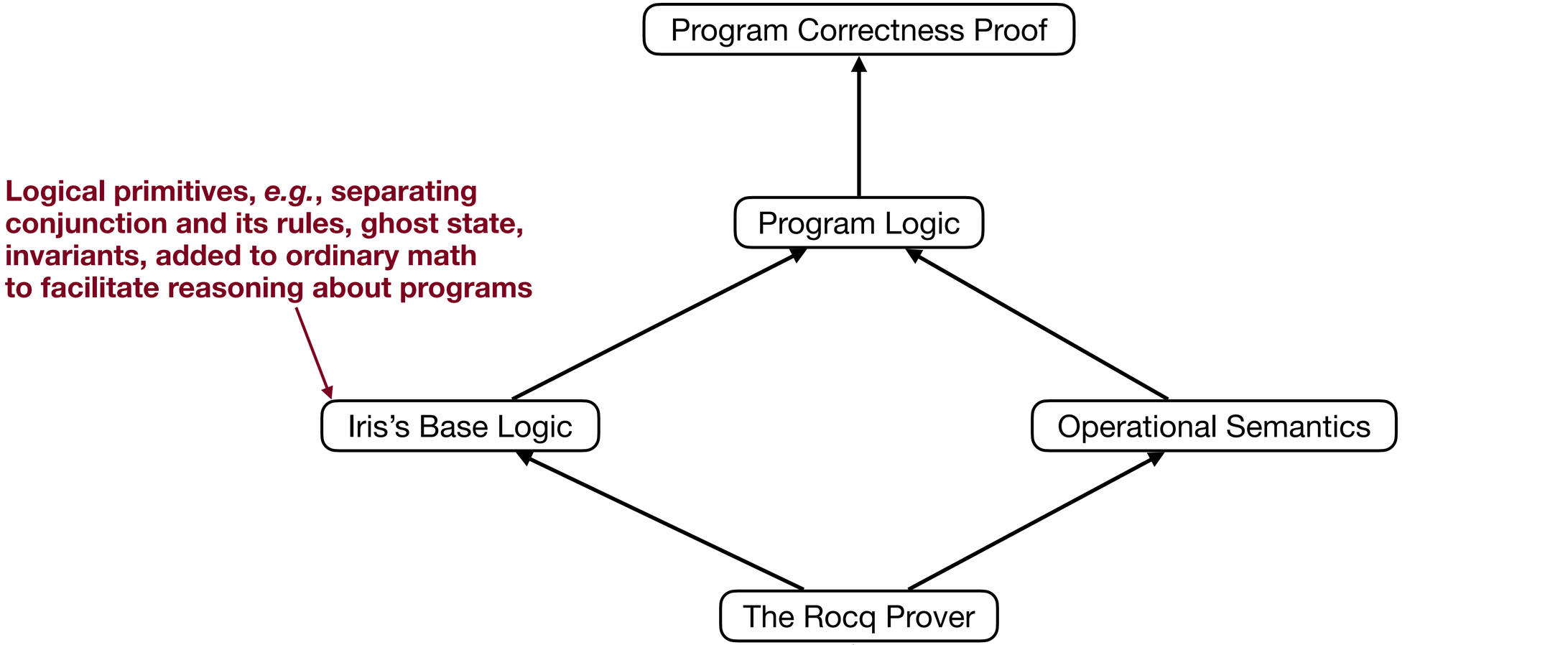
**Loading from memory location $\ell$**

**Answer: Separation Logic's $\ell \hookrightarrow v$**

# Higher-Order Concurrent Separation Logic

## The Iris Framework

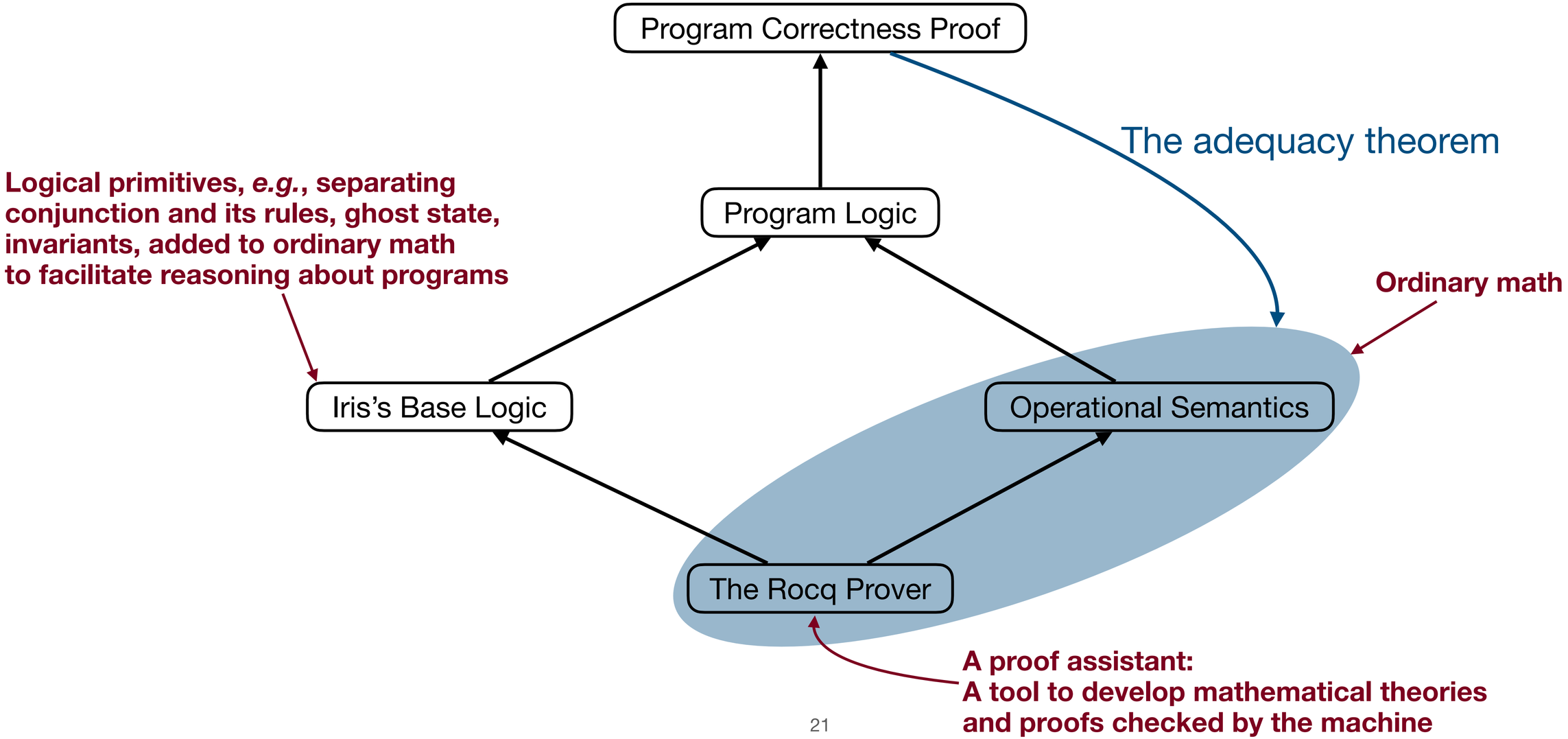
# The Iris Framework



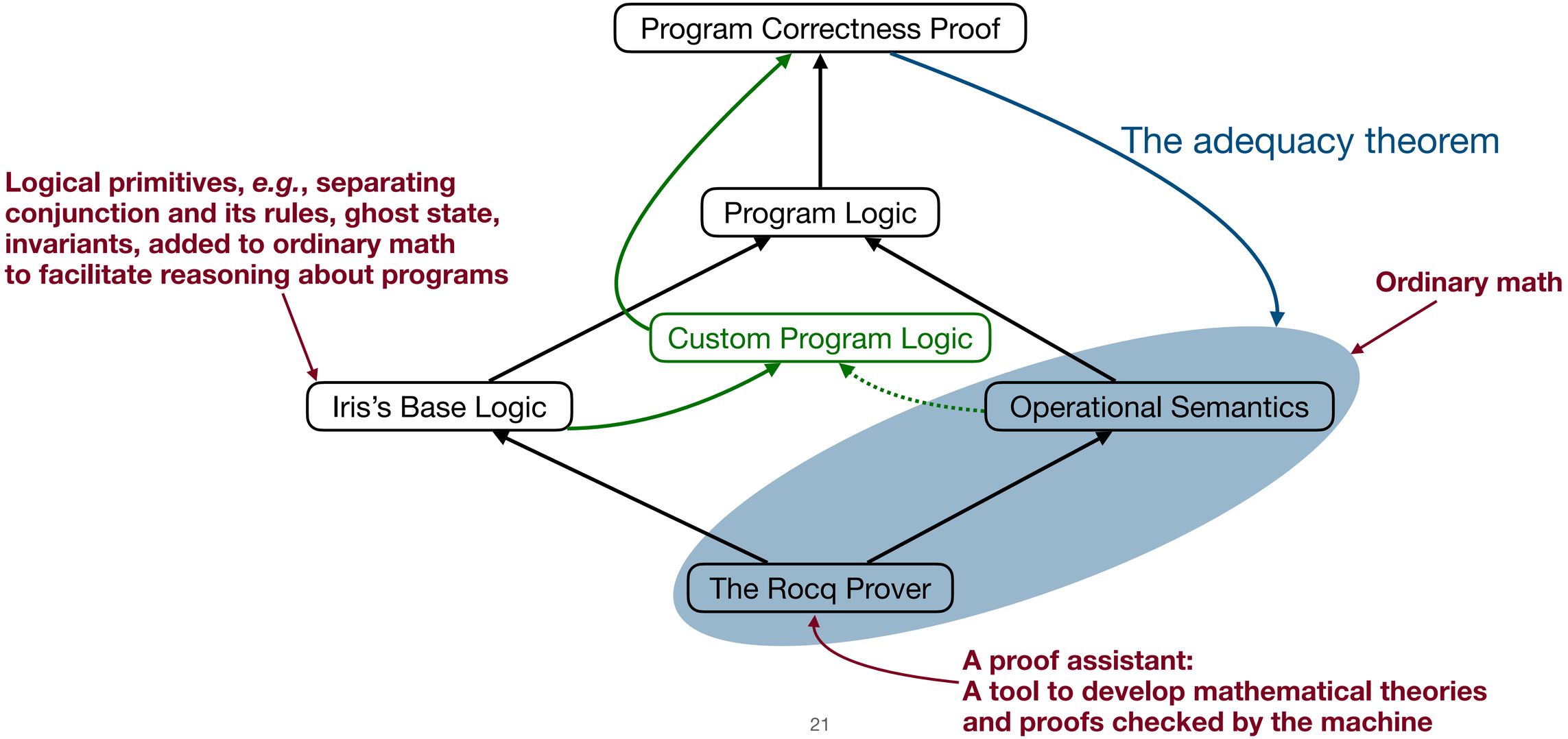

**Logical primitives, *e.g.*, separating conjunction and its rules, ghost state, invariants, added to ordinary math to facilitate reasoning about programs**

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

# The Iris Framework



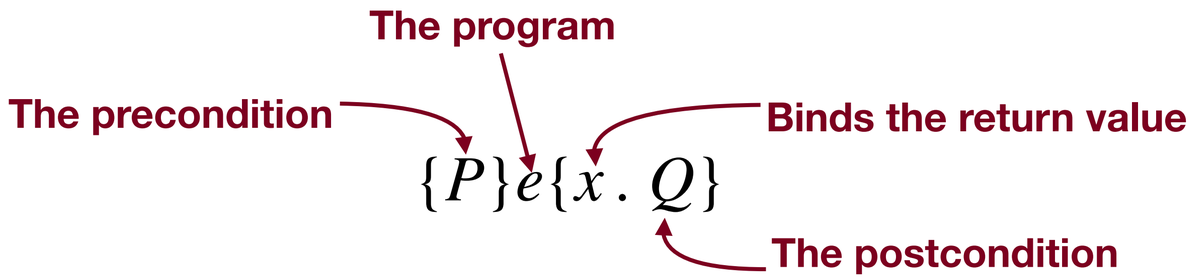
# The Iris Framework



# Iris's Program Logic

## A Higher-Order Separation Logic

**A Hoare-style logic:**




**Examples:**

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

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

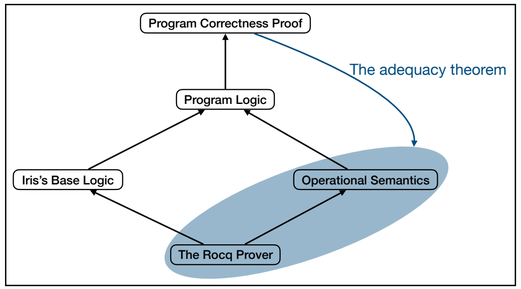
# The Adequacy Theorem

**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.