Scalable Foundations for Verified Systems Programming

Derek Dreyer
Max Planck Institute for Software Systems
(MPI-SWS)

Huawei Software Summit
July 7, 2022


Heartbleed: Hundreds of thousands of servers at risk from catastrophic bug
EternalBlue: A retrospective on one of the biggest Windows exploits ever
Google Reveals 5 New ‘High’ Rated Vulnerabilities In Chrome
Update now! Mozilla fixes security vulnerabilities in Firefox 94

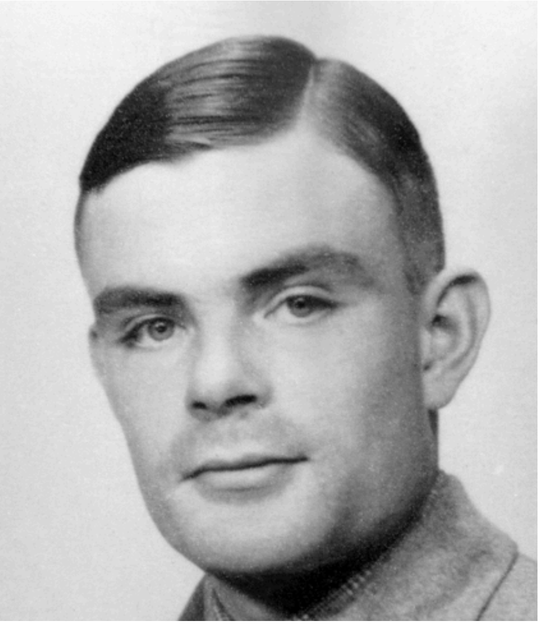
"Program testing can be used to show the presence of bugs, **but never to show their absence**!"

— Dijkstra (1970)

Goal of Program Verification:

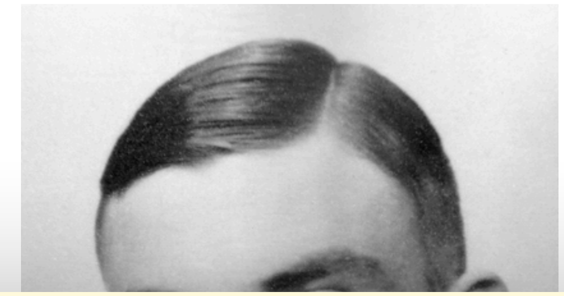
Build tools to establish rigorously that programs behave correctly in <u>all</u> executions.

Why is Verification Hard?



- Turing (1936): Halting problem (whether a program terminates) is **undecidable**
- Rice (1953): All non-trivial semantic (I/O) properties of programs are **undecidable** by reduction from the halting problem
- So there is **no complete method** for automatically verifying programs **in general**

Why is Verification Hard?



- Turing (1936): Halting problem

But we can still build verification tools that are **sound** and **practically useful**!

by reduction from the halting problem

- So there is **no complete method** for automatically verifying programs **in general**

Two compositional approaches to program verification

$\Gamma \vdash e : \tau$

$\{P\}\ e\ \{Q\}$

Two compositional approaches to program verification

$\Gamma \vdash e : \tau$

$\{P\}\ e\ \{Q\}$

Typing judgment

Hoare triple

Two compositional approaches to program verification

$\Gamma \vdash e : \tau$ $\{P\}\ e\ \{Q\}$

Modular program component

Two compositional approaches to program verification

$\Gamma \vdash e : \tau$ $\qquad\qquad$ $\{P\}\ e\ \{Q\}$

Assumptions

Two compositional approaches to program verification

$\Gamma \vdash e : \tau$

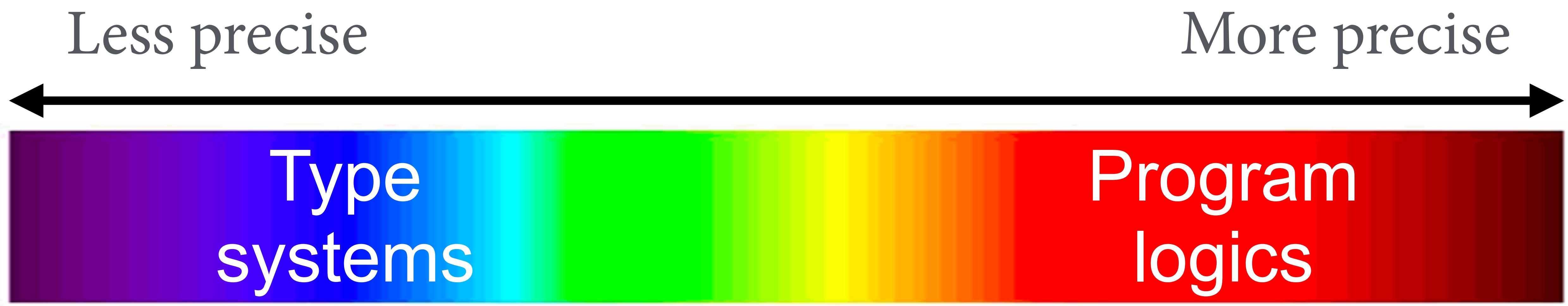
$\{P\}\ e\ \{Q\}$

Results

Two compositional approaches to program verification

$$\Gamma \vdash e : \tau \qquad\qquad \{P\}\ e\ \{Q\}$$

Two compositional approaches to program verification



$$\Gamma \vdash e : \tau \qquad\qquad \{P\}\ e\ \{Q\}$$

Type systems	Program logics
✔ Fully automated	✘ Semi-automated or manual
✔ Widely used by programmers	✘ Mostly used by verif. experts
✘ Shallow specs (e.g. safety)	✔ Deep specs (e.g. correctness)
✘ Restricts coding style	✔ Does not restrict coding style

Two compositional approaches to program verification

How can we marry the benefits of type systems & program logics together?

$\Gamma \vdash e : \tau$

- ✓ Fully automated
- ✓ Widely used by programmers
- ✗ Shallow specs (e.g. safety)
- ✗ Restricts coding style

$\{P\}\ e\ \{Q\}$

- ✗ Semi-automated or manual
- ✗ Mostly used by verif. experts
- ✓ Deep specs (e.g. correctness)
- ✓ Does not restrict coding style

Type systems

Program logics

extensibility

Type systems

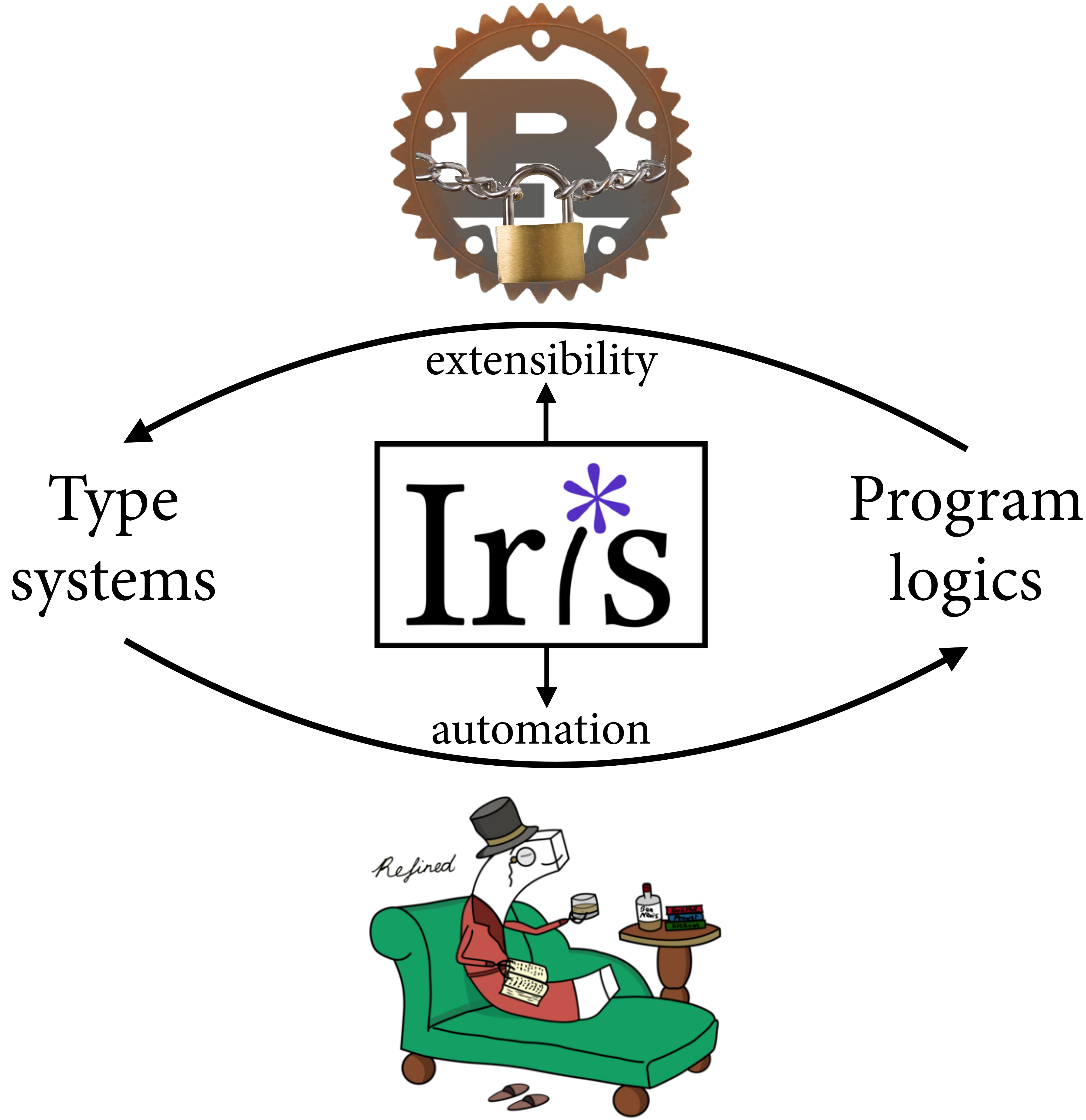
Program logics

extensibility

Type systems

Program logics

automation


extensibility
Type systems
Iris
Program logics
automation
Refined

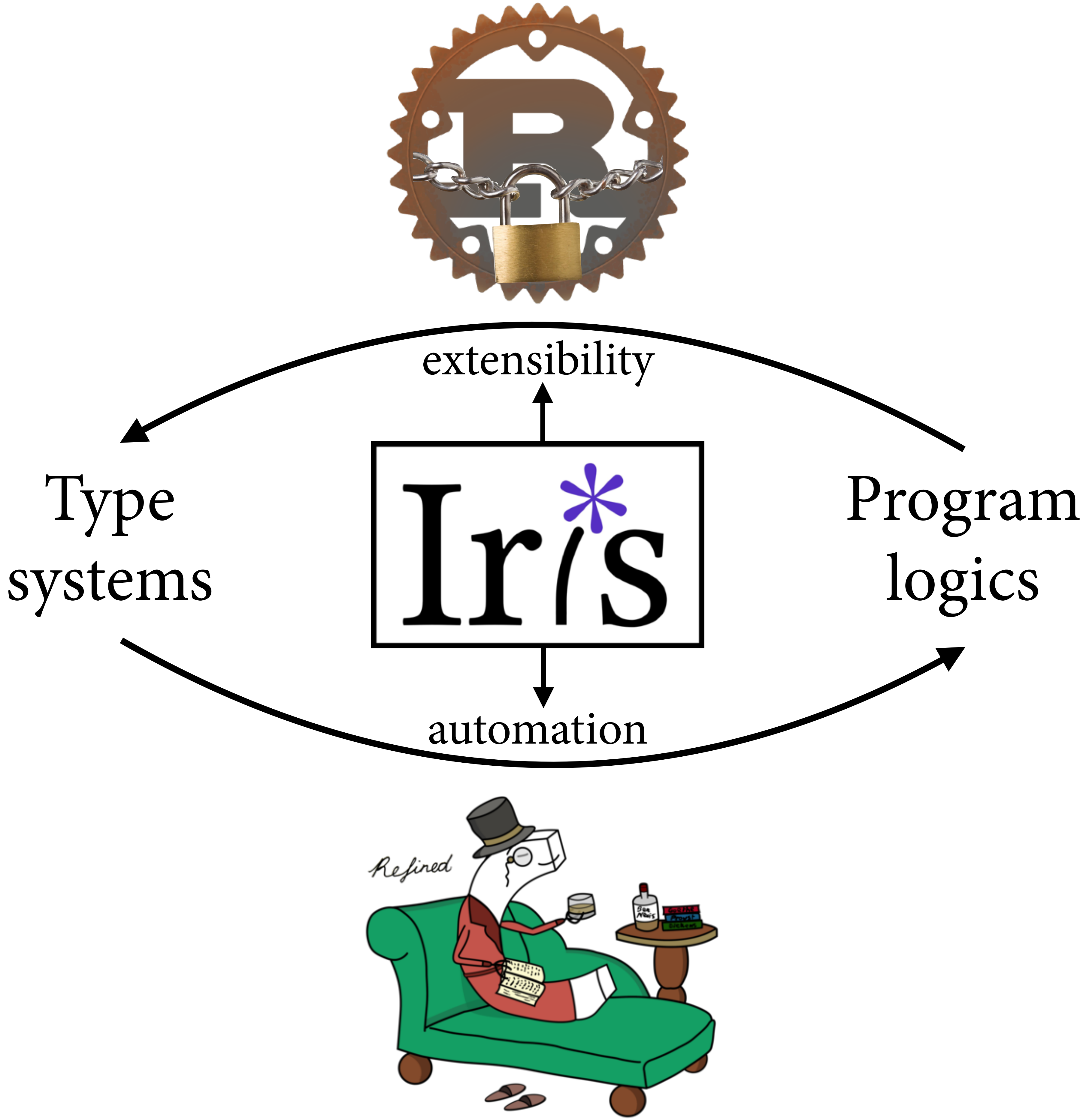
extensibility

Type systems

Iris

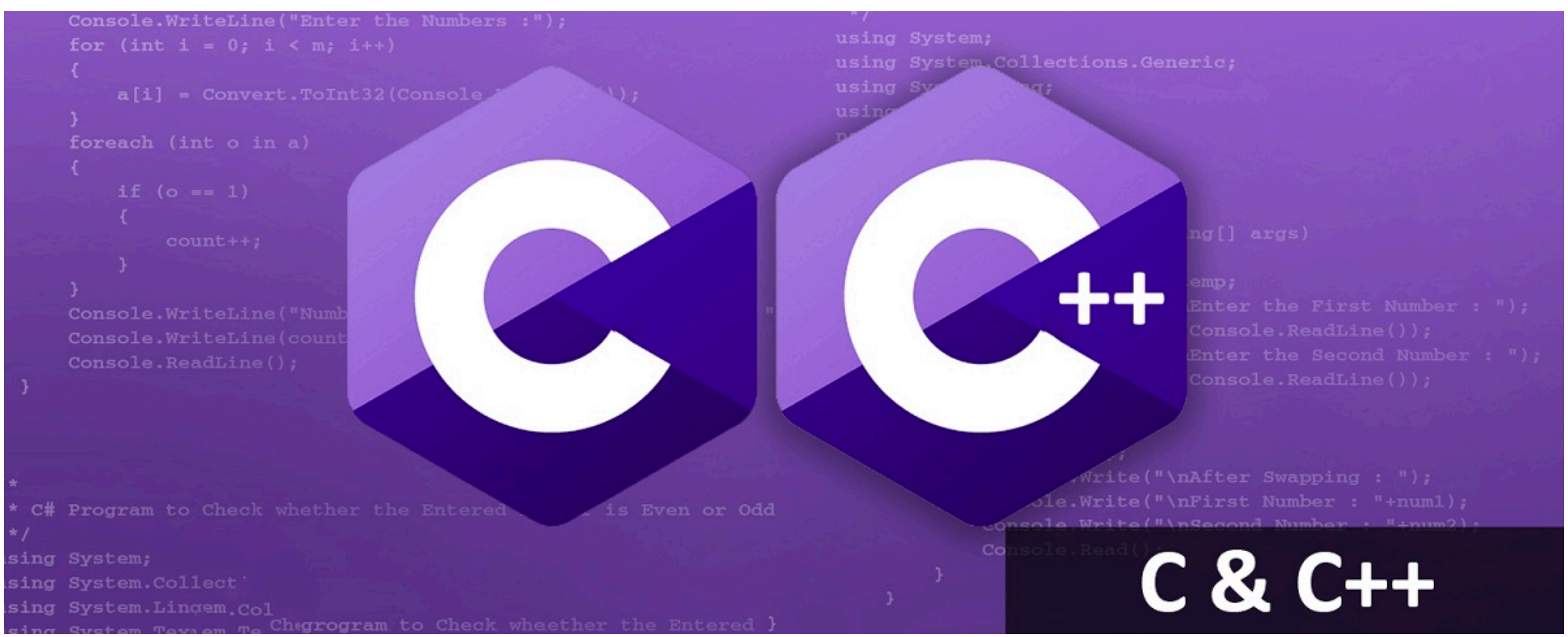
Program logics

automation


extensibility
Type systems
Iris
Program logics
automation
Refined

A Longstanding Problem

- Many core systems applications require low-level control over memory/resources
- Such applications are typically written in



A Longstanding Problem

- Many core systems applications require low-level control over memory/resources
- Such applications are typically written in

from Google Security Blog

An update on Memory Safety in Chrome

September 21, 2021

> Last year, we showed that more than 70% of our severe security bugs are memory safety problems. That is, mistakes with pointers in the C or C++ languages which cause memory to be misinterpreted.

---

from Microsoft Security Response Center

We need a safer systems programming language

Security Research & Defense / By MSRC Team / July 18, 2019 / Memory Safety, Rust, Safe Systems Programming Languages, Secure Development

> As was pointed out in our previous post, the root cause of approximately 70% of security vulnerabilities that Microsoft fixes and assigns a CVE (Common Vulnerabilities and Exposures) are due to memory safety issues. This is despite mitigations including intense code review, training, static analysis, and more.

from Google Security Blog

An update on Memory Safety in Chrome

September 21, 2021

> Last year, we showed that more than 70% of our severe security bugs are memory safety problems. That is, mistakes with pointers in the C or C++ languages which cause memory to be misinterpreted.

---

from Microsoft Security Response Center

We need a safer systems programming language

Security Research & Defense / By MSRC Team / July 18, 2019 / Memory Safety, Rust, Safe Systems Programming Languages, Secure Development

> As was pointed out in our previous post, the root cause of approximately 70% of security vulnerabilities that Microsoft fixes and assigns a CVE (Common Vulnerabilities and Exposures) are due to memory safety issues. This is despite mitigations including intense code review, training, static analysis, and more.

from Google Security Blog

An update on Memory Safety in Chrome

September 21, 2021

Last year, we showed that more than 70% of our severe security bugs are memory safety problems. That is, mistakes with pointers in the C or C++ languages which cause memory to be misinterpreted.

---

from Microsoft Security Response Center

We need a safer systems programming language

Security Research & Defense / By MSRC Team / July 18, 2019 / Memory Safety, Rust, Safe Systems Programming Languages, Secure Development

As was pointed out in our previous post, the root cause of approximately 70% of security vulnerabilities that Microsoft fixes and assigns a CVE (Common Vulnerabilities and Exposures) are due to memory safety issues. This is despite mitigations including intense code review, training, static analysis, and more.

from Google Security Blog

An update on Memory Safety in Chrome

September 21, 2021

Last year, we showed that more than 70% of our severe security bugs are memory safety problems. That is, mistakes with pointers in the C or C++ languages which cause memory to be misinterpreted.

from Microsoft Security Response Center

We need a safer systems programming language

Security Research & Defense / By MSRC Team / July 18, 2019 / Memory Safety, Rust, Safe Systems Programming Languages, Secure Development

As was pointed out in our previous post, the root cause of approximately 70% of security vulnerabilities that Microsoft fixes and assigns a CVE (Common Vulnerabilities and Exposures) are due to memory safety issues. This is despite mitigations including intense code review, training, static analysis, and more.

Rust:

The Future of Safe Systems Programming?

In development since 2010, with 1.0 release in 2015

- Mozilla used Rust to build Servo, a next-gen browser engine, later incorporated into Firefox

Rust is the only "systems PL" to provide…

- Low-level control à la modern C++
- Strong safety guarantees
- Industrial development and backing

Many major companies using Rust in production

- In 2021, the **Rust Foundation** was formed, incl. Amazon, Google, Huawei, Meta, Microsoft, Mozilla

Rust:

The Future of Safe Systems Programming?

In development since 2010, with 1.0 release in 2015

- Mozilla used Rust to build Servo, a next-gen browser engine, later incorporated into Firefox

The "safety" of Rust is central to its promise.

But how do we know Rust is safe?

Many major companies using Rust in production

- In 2021, the **Rust Foundation** was formed, incl. Amazon, Google, Huawei, Meta, Microsoft, Mozilla

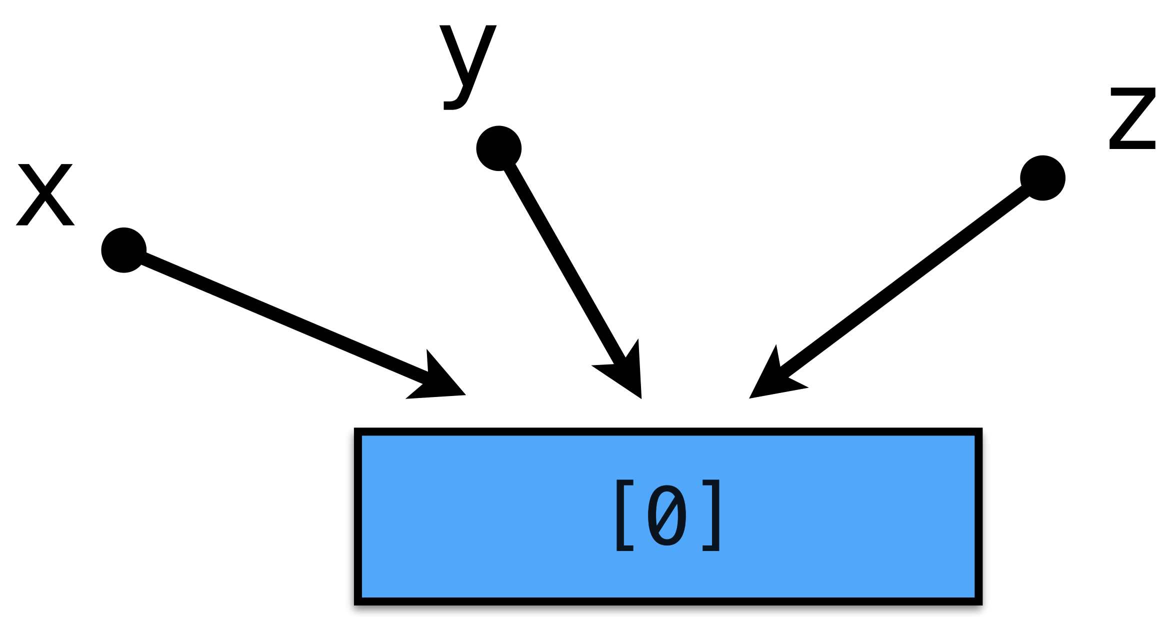
Core Idea of Rust

~~**Mutation**~~

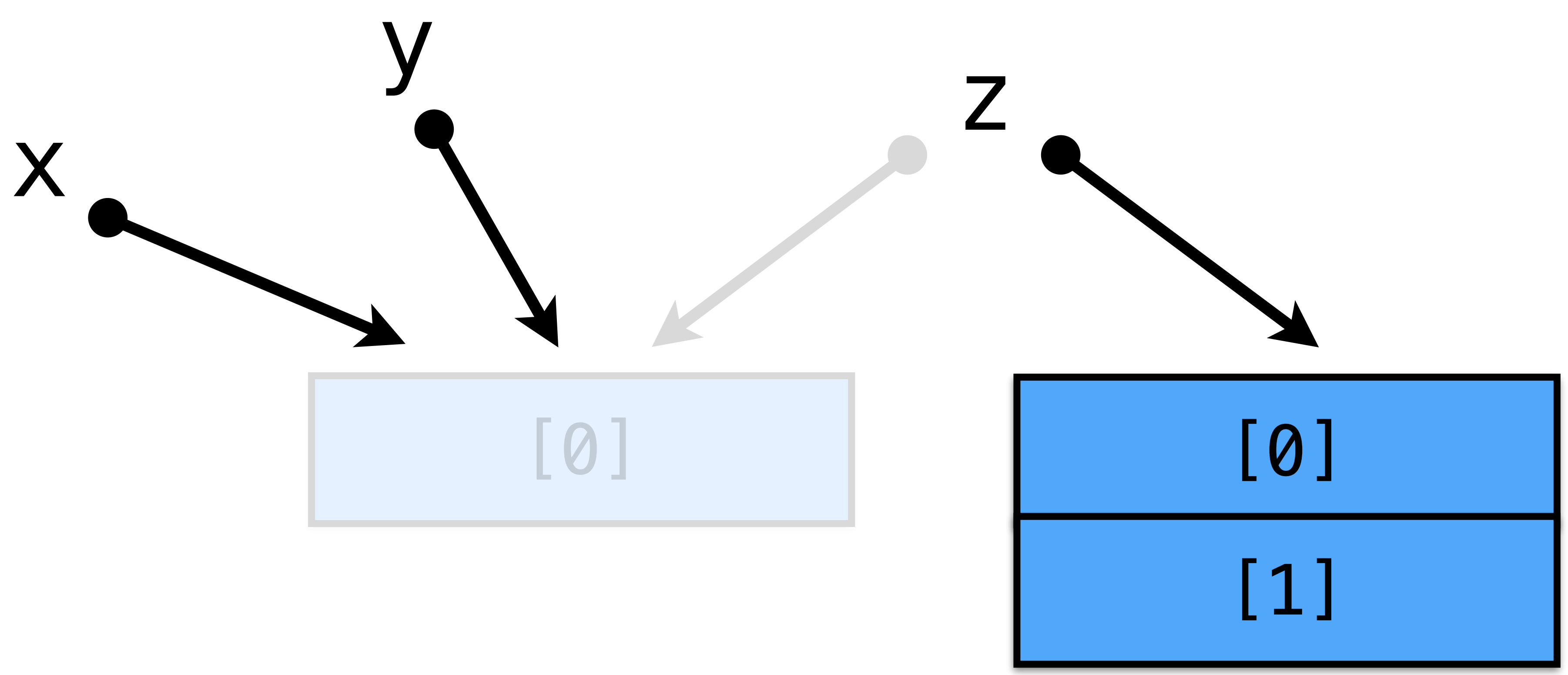
~~**+**~~

~~**Aliasing**~~

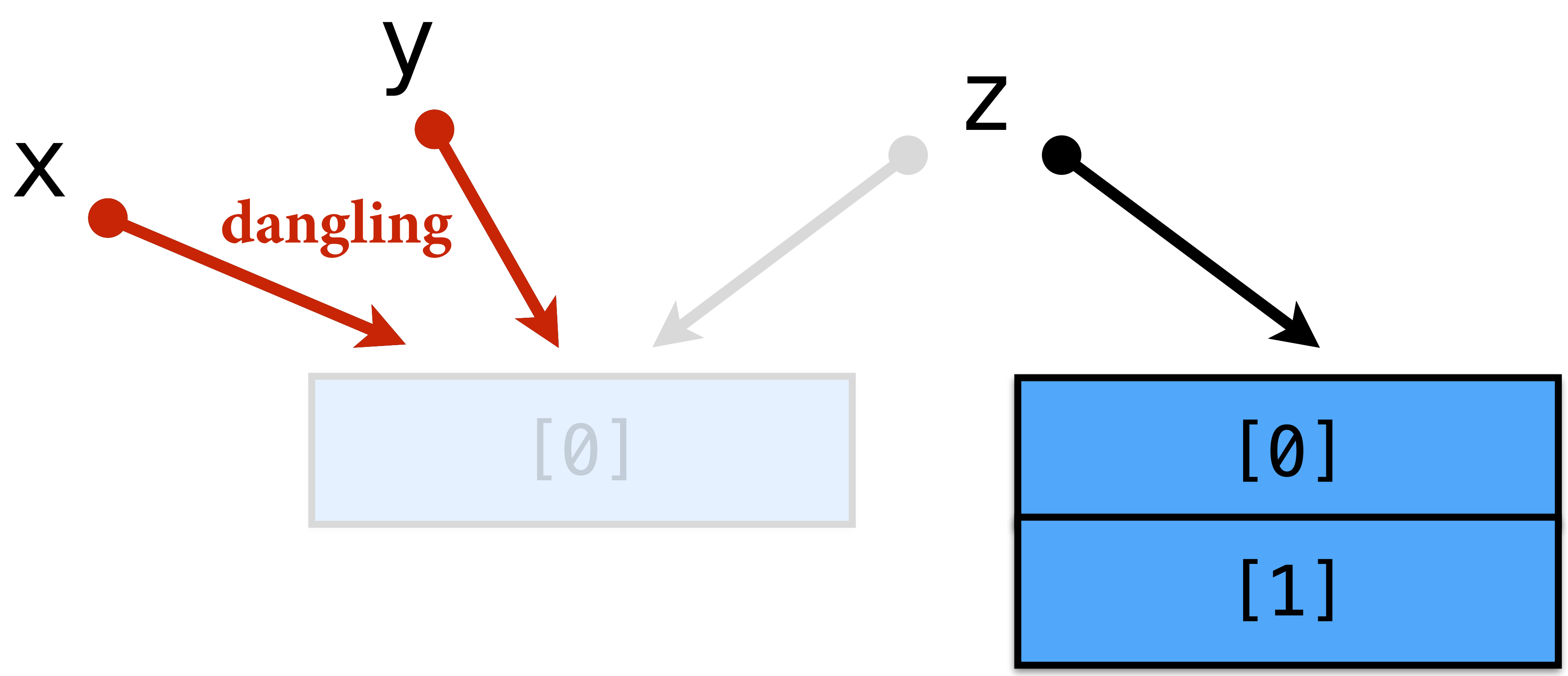
Core Idea of Rust



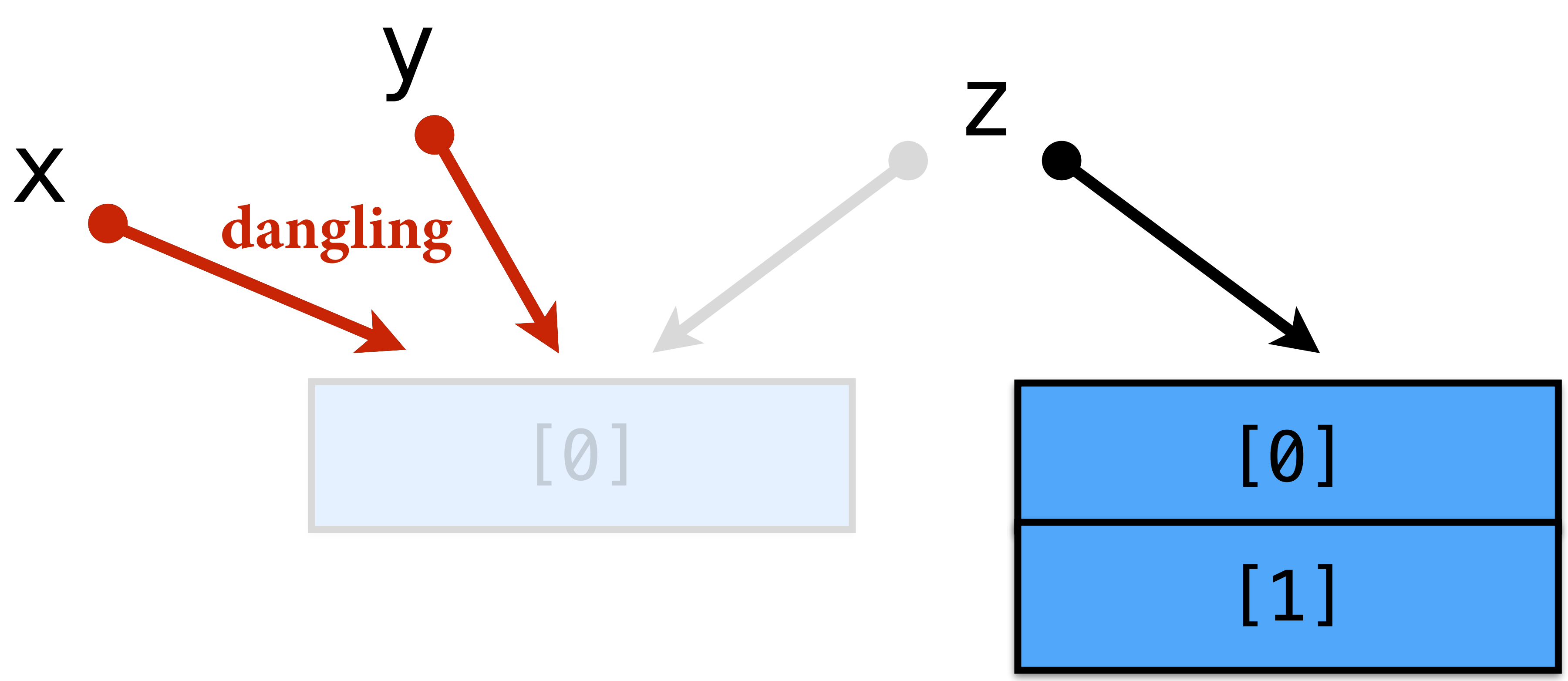
Core Idea of Rust



Core Idea of Rust



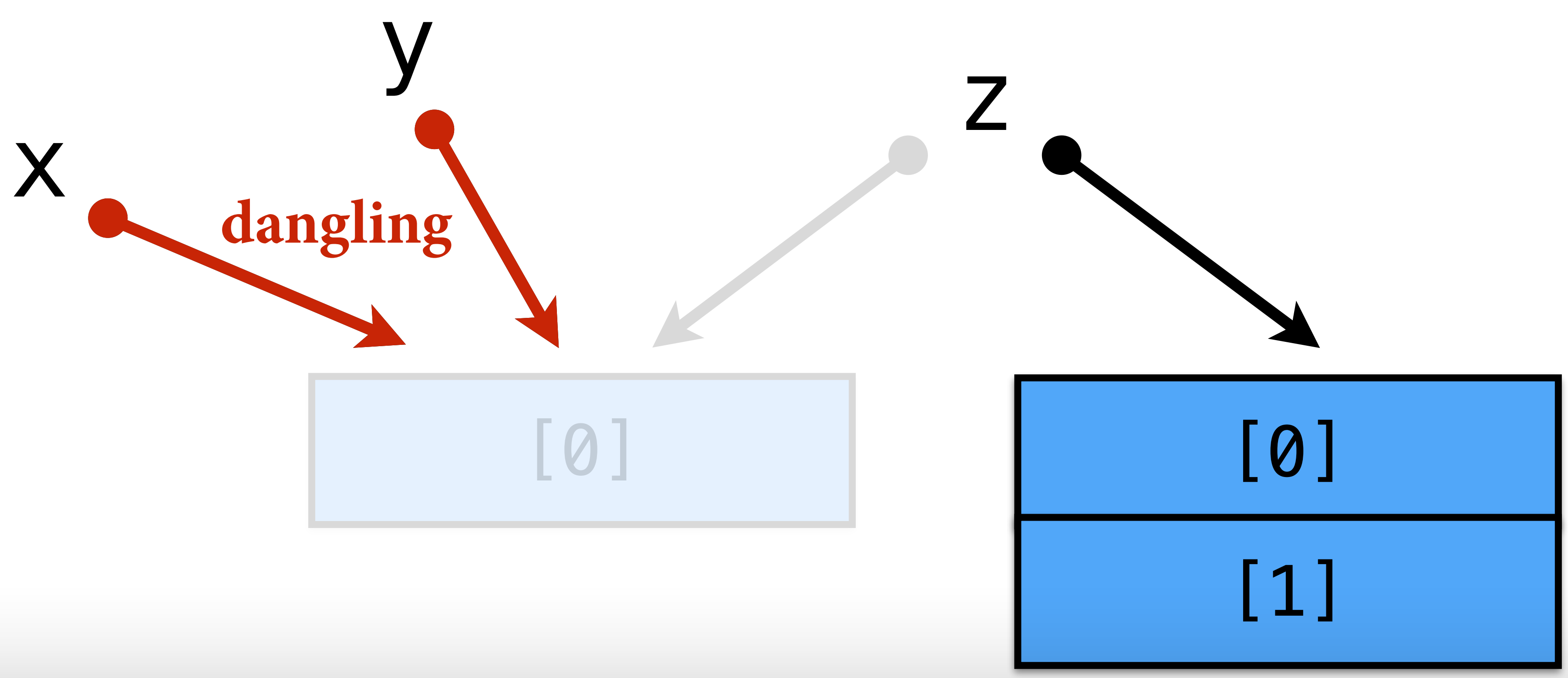
Core Idea of Rust



Unrestricted mutation and aliasing lead to:

- use-after-free errors (dangling references)
- data races
- iterator invalidation

Core Idea of Rust



Rust prevents all these errors using a sophisticated "ownership" type system

But sometimes you ***need*** **mutation + aliasing!**

Pointer-based data structures

- e.g. Doubly-linked lists

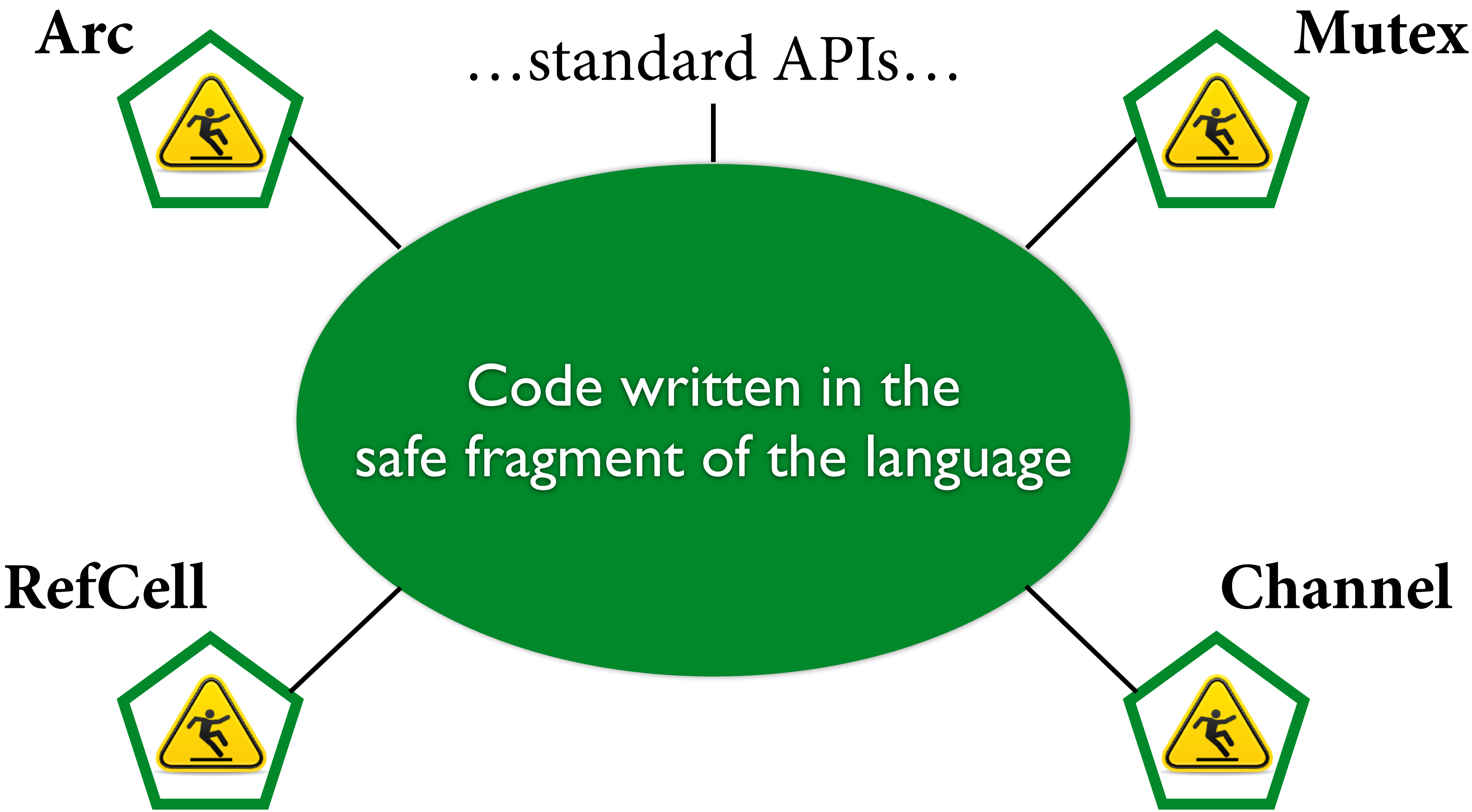
Synchronization mechanisms:

- e.g. Locks, channels, semaphores

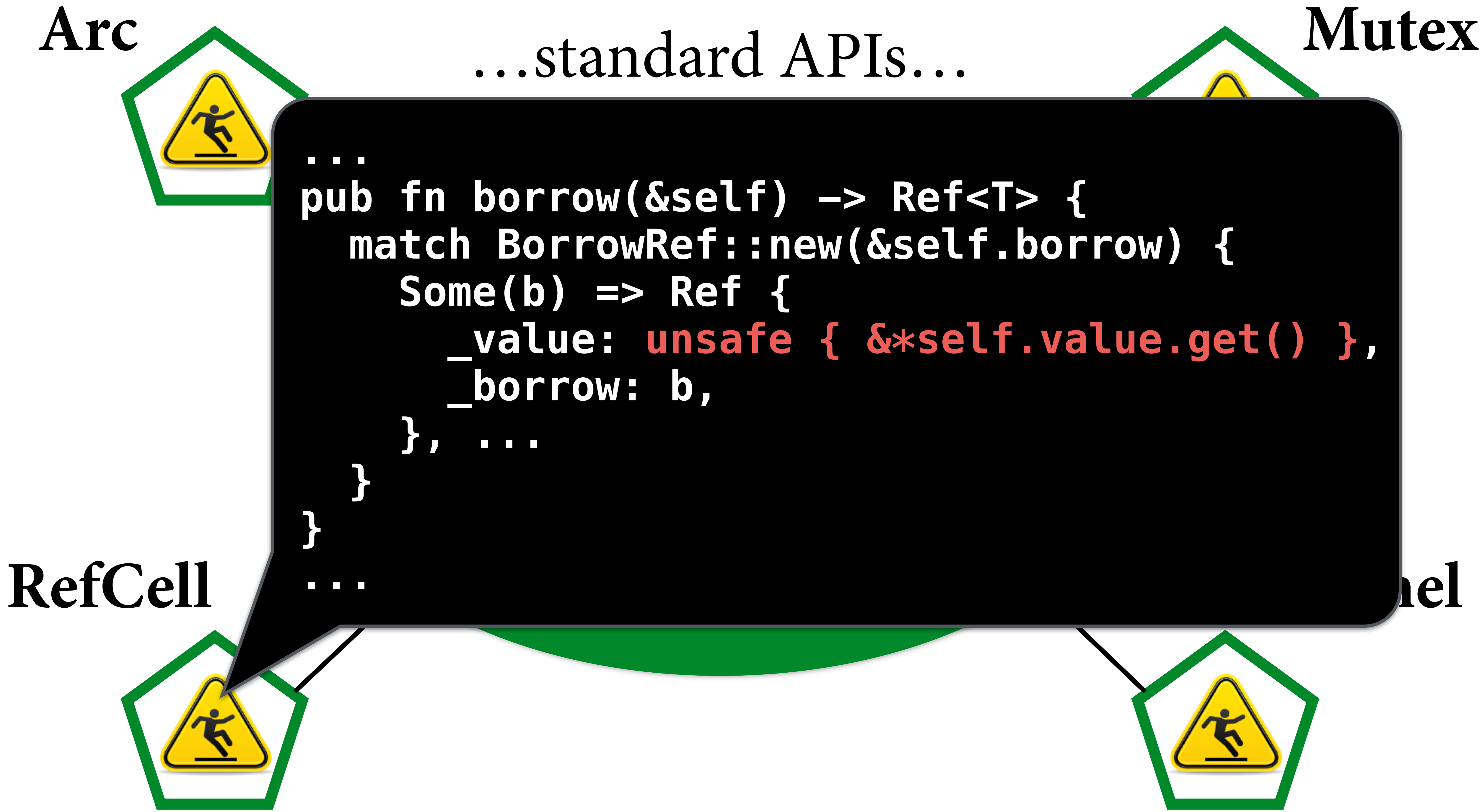
Memory management:

- e.g. Reference counting

The Reality of Rust

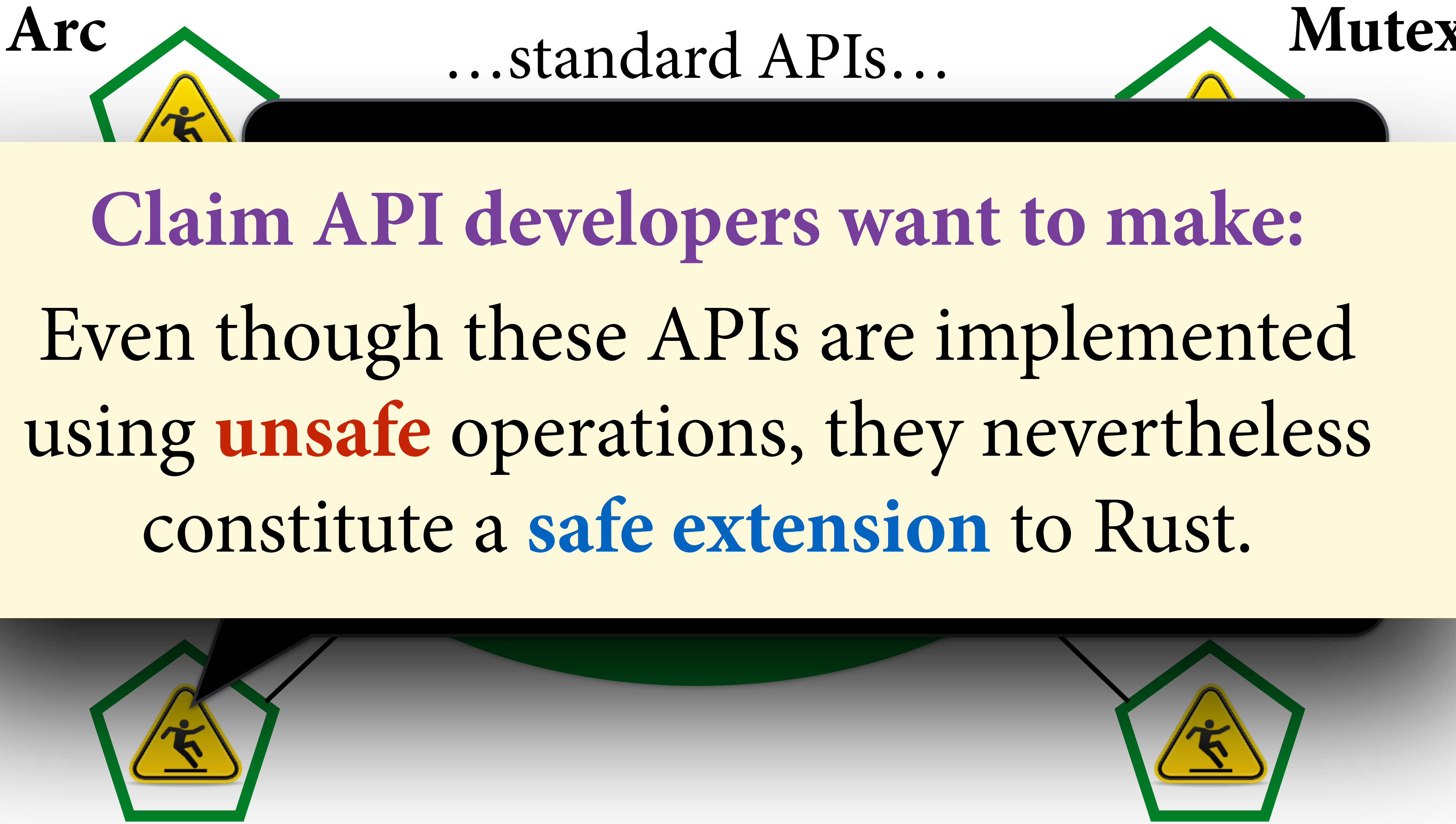


The Reality of Rust



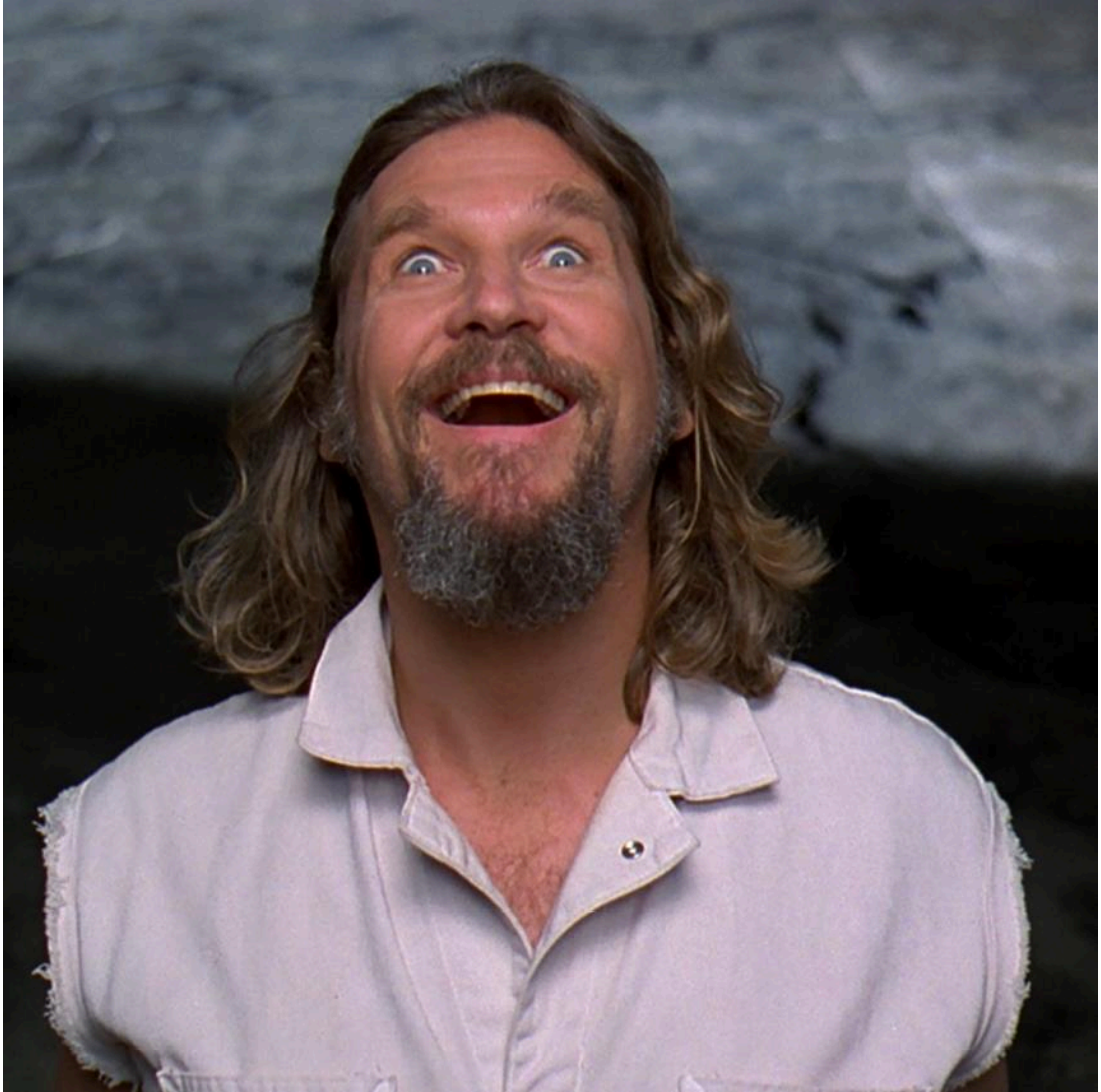

```
...
pub fn borrow(&self) -> Ref<T> {
  match BorrowRef::new(&self.borrow) {
    Some(b) => Ref {
      _value: unsafe { &*self.value.get() },
      _borrow: b,
    }, ...
  }
}
...
```

The Reality of Rust



Claim API developers want to make:

Even though these APIs are implemented using **unsafe** operations, they nevertheless constitute a **safe extension** to Rust.



RustBelt

European Research Council

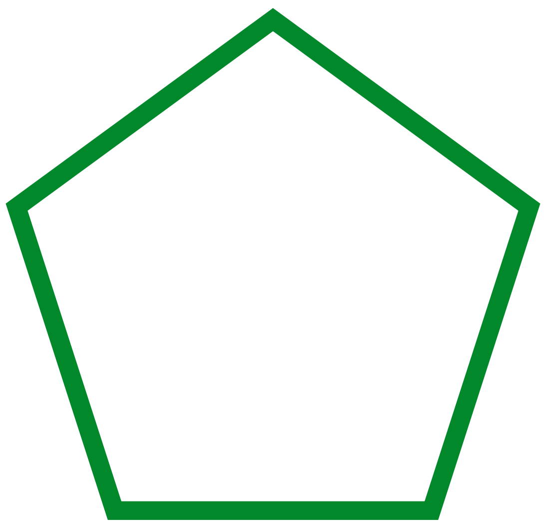
Goal: **Develop 1st formal foundations for Rust**

- Use these foundations to verify the safety of the Rust core type system and std APIs
- Give Rust developers the tools they need to safely evolve the language

Key Challenge

- Standard **"syntactic safety"** approach of Wright and Felleisen (1994) **will not work for Rust!**
  - Not applicable to programs with unsafe code
- Need to generalize to **semantic safety**
  - An API is semantically safe if no (well-typed) client of it will ever encounter unsafe behavior

Semantic Safety



API

$\rightsquigarrow$

semantic model

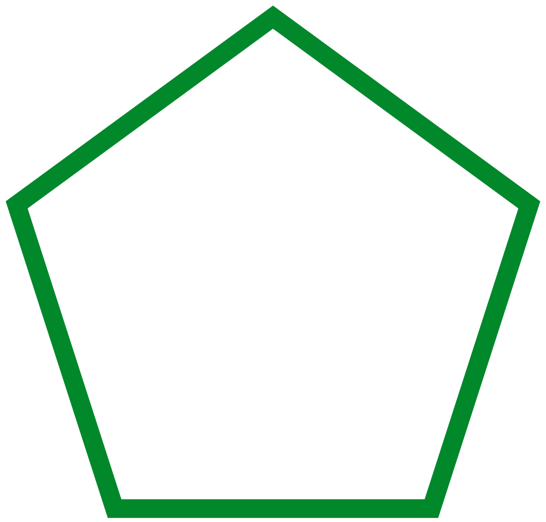
Safety contract

$\Dashv$

logical satisfaction

API implementation

Semantic Safety



API

$\rightsquigarrow$

semantic model

Safety contract

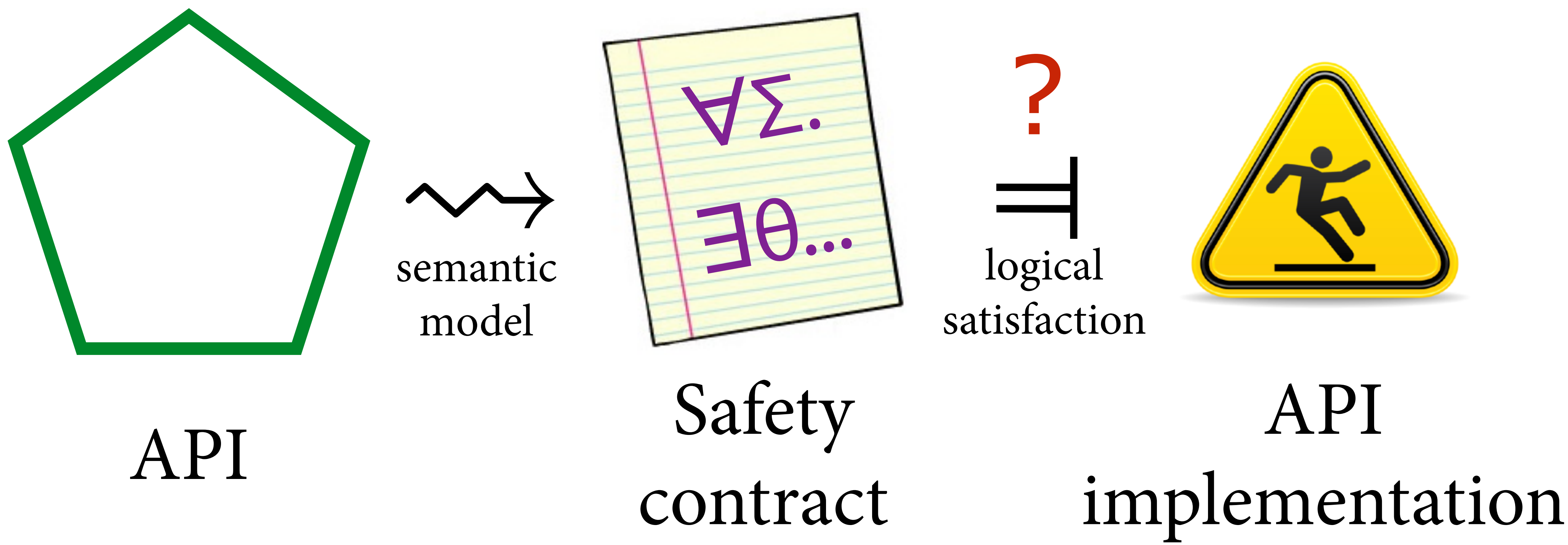
$\Dashv$

logical satisfaction

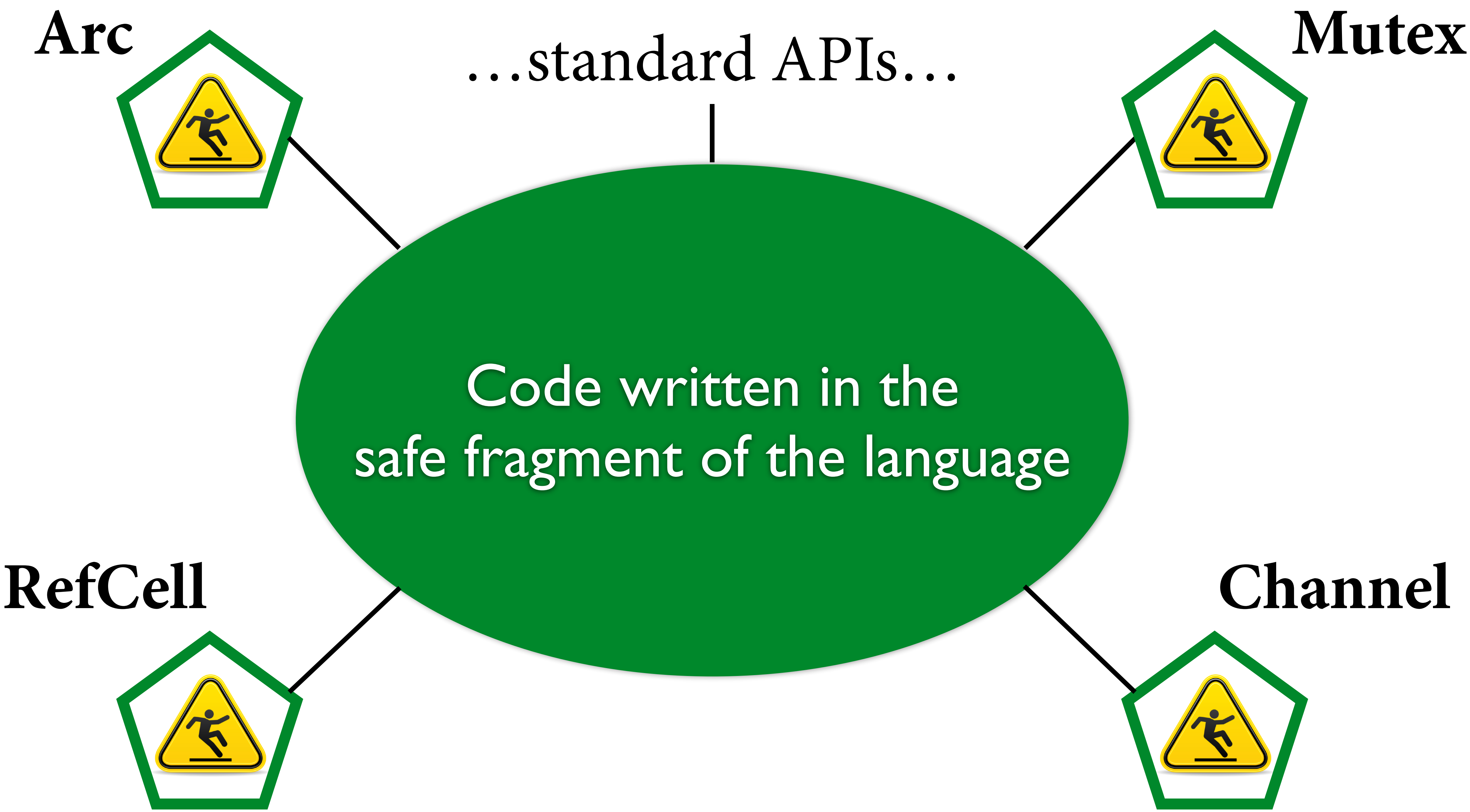
Safe fragment

API implementation

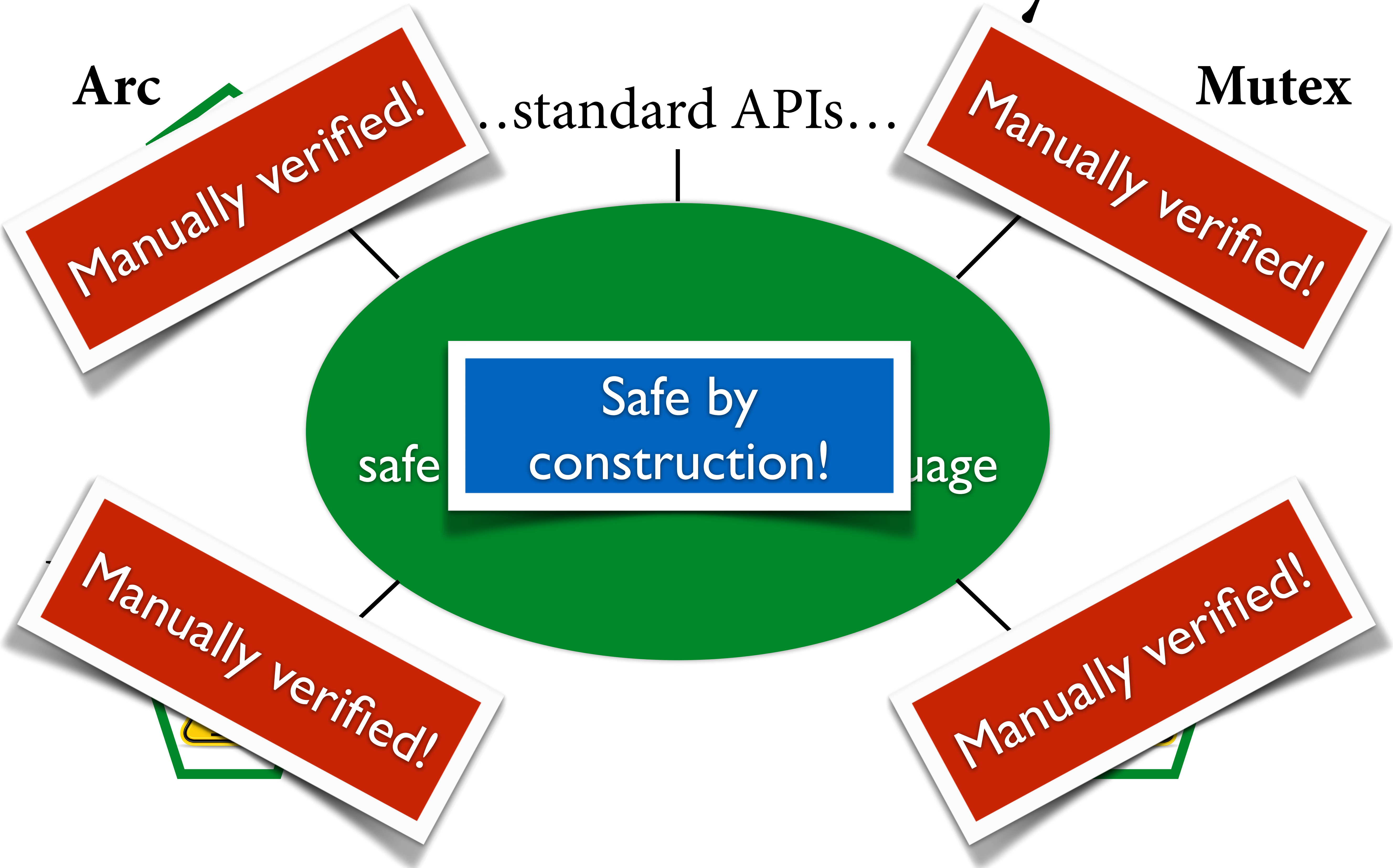
Semantic Safety



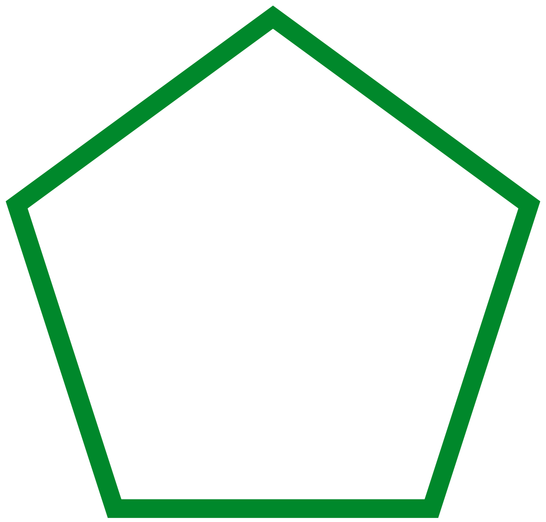
Semantic Safety



Semantic Safety



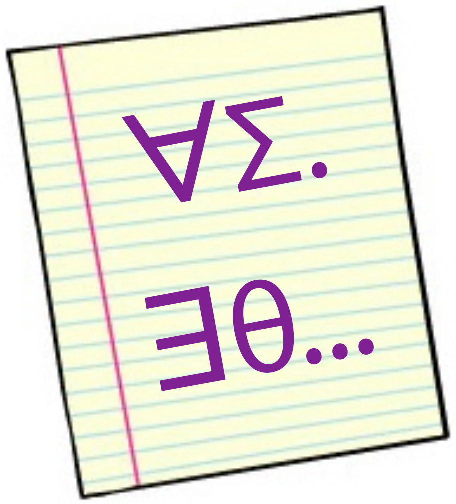
Semantic Safety



API

$\rightsquigarrow$

semantic model



Safety contract

$=\!\!|$

logical satisfaction

API implementation

RustBelt [POPL'18, POPL'20]:

Verifying semantic safety for Rust

Heart of the Problem

How do we define our semantic model of Rust?

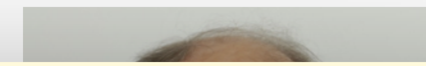
Key Prior Work on Semantic Models

- **Milner (1978).** Used a "**logical-relations**" model based on **denotational** semantics. But didn't scale to richer type systems.

- **Appel-McAllester (2001).** Introduced a "**step-indexed**" logical-relations model of recursive types using **operational** semantics.

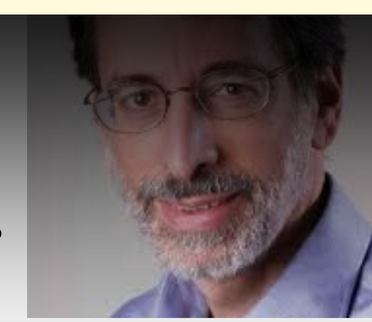
- **Ahmed (2004).** Scaled step-indexed model to handle **higher-order state** using recursive Kripke structures. Landmark PhD dissertation.

Key Prior Work on Semantic Models



Ahmed's work was a major inspiration for me, but there was a scalability problem…

"**step-indexed**" logical-relations model of recursive types using **operational** semantics.



- **Ahmed (2004).** Scaled step-indexed model to handle **higher-order state** using recursive Kripke structures. Landmark PhD dissertation.

Theorem 3.21 (Application)
If Γ is a type environment, e_1 and e_2 are (possibly open) terms, and τ_1 and τ_2 are types such that $\Gamma \vDash_M e_1 : \tau_1 \to \tau_2$ and $\Gamma \vDash_M e_2 : \tau_1$ then $\Gamma \vDash_M (e_1\, e_2) : \tau_2$.

PROOF: We must prove that under the premises of the theorem, for any $k \geq 0$, we have $\Gamma \vDash_M^k (e_1\, e_2) : \tau_2$. More specifically, for any σ and Ψ such that $\sigma :_{k,\Psi} \Gamma$ we must show $\sigma(e_1\, e_2) :_{k,\Psi} \tau_2$. From the premise $\Gamma \vDash_M e_1 : \tau_1 \to \tau_2$ we have $\sigma(e_1) :_{k,\Psi} \tau_1 \to \tau_2$. To show $\sigma(e_1\, e_2) :_{k,\Psi} \tau_2$, suppose $S :_k \Psi$ for some store S. Then, from $\sigma(e_1) :_{k,\Psi} \tau_1 \to \tau_2$ it follows that $(S, \sigma(e_1))$ is safe for k steps. Either $(S, \sigma(e_1))$ reduces for k steps without reaching a state (S', v_1) where v_1 is a value — in which case $(S, \sigma(e_1\, e_2))$ does not generate a value in less than k steps and hence $\sigma(e_1\, e_2) :_{k,\Psi} \tau_2$ (for any τ_2) — or the value v_1 is a lambda expression $\lambda x.e$. In the latter case, since $\sigma(e_1) :_{k,\Psi} \tau_1 \to \tau_2$ and $(S, \sigma(e_1)) \longmapsto_M^j (S', \lambda x.e)$, where $j < k$ and $\mathrm{irred}(S', \lambda x.e)$, it follows that there exists a Ψ' such that $(k, \Psi) \sqsubseteq (k-j, \Psi')$ and $S' :_{k-j} \Psi'$ and $\langle k-j, \Psi', \lambda x.e \rangle \in \tau_1 \to \tau_2$.

From $\sigma :_{k,\Psi} \Gamma$ it follows that $\sigma(x) :_{k,\Psi} \Gamma(x)$ for all variables $x \in \mathrm{dom}(\Gamma)$. A type environment Γ is a mapping from variables to types. Hence, since $(k, \Psi) \sqsubseteq (k-j, \Psi')$, it follows that $\sigma(x) :_{k-j,\Psi'} \Gamma(x)$ for all $x \in \mathrm{dom}(\Gamma)$ — that is, we have $\sigma :_{k-j,\Psi'} \Gamma$. Now, from premise $\Gamma \vDash_M e_2 : \tau_1$, since $k - j \geq 0$, $S' :_{k-j} \Psi'$, and $\sigma :_{k-j,\Psi'} \Gamma$, we have $\sigma(e_2) :_{k-j,\Psi'} \tau_1$. It follows that $(S', \sigma(e_2))$ is safe for $k-j$ steps, i.e., either $(S', \sigma(e_2))$ does not generate a value in fewer than $k-j$ steps — in which case, $(S, \sigma(e_1\, e_2))$ does not generate a value in fewer than k steps so we have $\sigma(e_1\, e_2) :_{k,\Psi} \tau_2$ (for any τ_2) — or $(S', \sigma(e_2)) \longmapsto_M^i (S'', v)$ where $i < k - j$. In the latter case, $(S, \sigma(e_1\, e_2)) \longmapsto_M^{j+i} (S'', (\lambda x.e)v)$ where $j + i < k$. Also, since $\sigma(e_2) :_{k-j,\Psi'} \tau_1$ and $(S', \sigma(e_2)) \longmapsto_M^i (S'', v)$ for $i < k - j$ and $\mathrm{irred}(S'', v)$, it follows that there exists a Ψ'' such that $(k-j, \Psi') \sqsubseteq (k-j-i, \Psi'')$, $S'' :_{k-j-i} \Psi''$, and $\langle k-j-i, \Psi'', v \rangle \in \tau_1$.

Pick memory typing $\Psi^* = \lfloor \Psi'' \rfloor_{k-j-i-1}$. Let $k^* = k-j-i-1$. Then the following information-forgetting state extension holds: $(k-j-i, \Psi'') \sqsubseteq (k^*, \Psi^*)$. Since $\langle k-j-i, \Psi'', v \rangle \in \tau_1$ and τ_1 is a type, we have $\langle k^*, \Psi^*, v \rangle \in \tau_1$. The definition of $\to$ then implies that $e[v/x] :_{k^*,\Psi^*} \tau_2$. But we now have $(S, \sigma(e_1\, e_2)) \longmapsto_M^{j+i+1} (S'', e[v/x])$, $(k, \Psi) \sqsubseteq (k^*, \Psi^*)$, $S'' :_{k^*} \Psi^*$, and $e[v/x] :_{k^*,\Psi^*} \tau_2$. By Definition 3.6 (Expr : Type), this means that if $(S'', e[v/x])$ generates a value in fewer than k^* steps then that value will be of type τ_2. Hence, we may conclude that $\sigma(e_1\, e_2) :_{k,\Psi} \tau_2$ as we wanted to show. $\square$

$e_1\, e_2)$) does not generate a value in fewer than k steps s
(for any τ_2) — or $(S', \sigma(e_2)) \longmapsto^{i}_{M} (S'', v)$ where $i < k$
$', \sigma(e_1\, e_2)) \longmapsto^{j+i}_{M} (S'', (\lambda x.e)v)$ where $j + i < k$. Also,
$'', \sigma(e_2)) \longmapsto^{i}_{M} (S'', v)$ for $i < k - j$ and irred(S'', v), it fo
$''$ such that $(k - j, \Psi') \sqsubseteq (k - j - i, \Psi'')$, $S'' :_{k-j-i} \Psi''$,
.
ng $\Psi^* = \lfloor \Psi'' \rfloor_{k-j-i-1}$. Let $k^* = k - j - i - 1$. Then the fo
tting state extension holds: $(k - j - i, \Psi'') \sqsubseteq (k^*, \Psi^*)$. S
$_1$ and τ_1 is a type, we have $\langle k^*, \Psi^*, v \rangle \in \tau_1$. The defin
hat $e[v/x] :_{k^*,\Psi^*} \tau_2$. But we now have $(S, \sigma(e_1\, e_2)) \longmapsto$
(k^*, Ψ^*), $S'' :_{k^*} \Psi^*$, and $e[v/x] :_{k^*,\Psi^*} \tau_2$. By Definitio
eans that if $(S'', e[v/x])$ generates a value in fewer than k^*
be of type τ_2. Hence, we may conclude that $\sigma(e_1\, e_2) :_{k,\Psi}$

$e_1\, e_2))$ does not generate a value in fewer than k steps s

(for any τ_2) — or $(S', \sigma(e_2)) \longmapsto^{i}_{M} (S'', v)$ where $i < k$

$', \sigma(e_1\, e_2)) \longmapsto^{j+i}_{M} (S'', (\lambda x.e)v)$ where $j + i < k$. Also,

$'', \sigma(e_2)) \longmapsto^{i}$ irred(S'', v), it fo

$''$ such tha $), S'' :_{k-j-i} \Psi''$,

Extraneous low-level details that obscure the proof idea!

.

ng $\Psi^* = \lfloor \Psi'' \rfloor_{k-j-i-1}$. $\boxed{\text{Let } k^* = k - j - i - 1.}$ Then the fo

tting state extension holds: $(k - j - i, \Psi'') \sqsubseteq (k^*, \Psi^*)$.

$_1$ and τ_1 is a type, we have $\langle k^*, \Psi^*, v \rangle \in \tau_1$. The defin

at $e[v/x] :_{k^*, \Psi^*} \tau_2$. But we now have $(S, \sigma(e_1\, e_2)) \longmapsto$

(k^*, Ψ^*), $\boxed{S'' :_{k^*} \Psi^*}$, and $e[v/x] :_{k^*, \Psi^*} \tau_2$. By Definitio

eans that if $(S'', e[v/x])$ generates a value in fewer than k^*

pe of type τ_2. Hence, we may conclude that $\sigma(e_1\, e_2) :_{k, \Psi}$

(e_2)) does not generate a value in fewer than k steps

$e_1\, e_2$)) does not generate a value in fewer than k steps s

(for any τ_2) — or $(S', \sigma(e_2)) \longmapsto^{i}_{M} (S'', v)$ where $i < k$

$', \sigma(e_1$ … Also,

$'', \sigma(e$ … , it fo

$''$ suc … $_i$ Ψ'',

.

ng Ψ^* … the fo

tting … Ψ^*). S

$_1$ and … defin

at $e[$ …) $\longmapsto$

$(k^*,$ … initio

eans t … an k^*

e of type τ_2. Hence, we may conclude that $\sigma(e_1\, e_2) :_{k,\Psi}$

Theorem 3.21 (Application)
If Γ is a type environment, e_1 and e_2 are (possibly open) terms, and τ_1 and τ_2 are types such that $\Gamma \models_M e_1 : \tau_1 \to \tau_2$ and $\Gamma \models_M e_2 : \tau_1$ then $\Gamma \models_M (e_1\, e_2) : \tau_2$.

PROOF: We must prove that under the premises of the theorem, for any $k \geq 0$, we have $\Gamma \models_M^k (e_1\, e_2) : \tau_2$. More specifically, for any σ and Ψ such that $\sigma :_{k,\Psi} \Gamma$ we must show $\sigma(e_1\, e_2) :_{k,\Psi} \tau_2$. From the premise $\Gamma \models_M e_1 : \tau_1 \to \tau_2$ we have $\sigma(e_1) :_{k,\Psi} \tau_1 \to \tau_2$. To show $\sigma(e_1\, e_2) :_{k,\Psi} \tau_2$, suppose $S :_k \Psi$ for some store S. Then, from $\sigma(e_1) :_{k,\Psi} \tau_1 \to \tau_2$ it follows that $(S, \sigma(e_1))$ is safe for k steps. Either $(S, \sigma(e_1))$ reduces for k steps without reaching a state (S', v_1) where v_1 is a value — in which case $(S, \sigma(e_1\, e_2))$ does not generate a value in less than k steps and hence $\sigma(e_1\, e_2) :_{k,\Psi} \tau_2$ (for any τ_2) — or the value v_1 is a lambda expression $\lambda x.e$. In the latter case, since $\sigma(e_1) :_{k,\Psi} \tau_1 \to \tau_2$ and $(S, \sigma(e_1)) \longmapsto_M^j (S', \lambda x.e)$, where $j < k$ and $\mathrm{irred}(S', \lambda x.e)$, it follows that there exists a Ψ' such that $(k, \Psi) \sqsubseteq (k-j, \Psi')$ and $S' :_{k-j} \Psi'$ and $\langle k-j, \Psi', \lambda x.e\rangle \in \tau_1 \to \tau_2$.

From $\sigma :_{k,\Psi} \Gamma$ it follows that $\sigma(x) :_{k,\Psi} \Gamma(x)$ for all variables $x \in \mathrm{dom}(\Gamma)$. A type environment Γ is a mapping from variables to types. Hence, since $(k, \Psi) \sqsubseteq (k-j, \Psi')$, it follows that $\sigma(x) :_{k-j,\Psi'} \Gamma(x)$ for all $x \in \mathrm{dom}(\Gamma)$ — that is, we have $\sigma :_{k-j,\Psi'} \Gamma$. Now, from premise $\Gamma \models_M e_2 : \tau_1$, since $k - j \geq 0$, $S' :_{k-j} \Psi'$, and $\sigma :_{k-j,\Psi'} \Gamma$, we have $\sigma(e_2) :_{k-j,\Psi'} \tau_1$. It follows that $(S', \sigma(e_2))$ is safe for $k-j$ steps, i.e., either $(S', \sigma(e_2))$ does not generate a value in fewer than $k-j$ steps — in which case, $(S, \sigma(e_1\, e_2))$ does not generate a value in fewer than k steps so we have $\sigma(e_1\, e_2) :_{k,\Psi} \tau_2$ (for any τ_2) — or $(S', \sigma(e_2)) \longmapsto_M^i (S'', v)$ where $i < k-j$. In the latter case, $(S, \sigma(e_1\, e_2)) \longmapsto_M^{j+i} (S'', (\lambda x.e) v)$ where $j + i < k$. Also, since $\sigma(e_2) :_{k-j,\Psi'} \tau_1$ and $(S', \sigma(e_2)) \longmapsto_M^i (S'', v)$ for $i < k-j$ and $\mathrm{irred}(S'', v)$, it follows that there exists a Ψ'' such that $(k-j, \Psi') \sqsubseteq (k-j-i, \Psi'')$, $S'' :_{k-j-i} \Psi''$, and $\langle k-j-i, \Psi'', v\rangle \in \tau_1$.

Pick memory typing $\Psi^* = \lfloor \Psi'' \rfloor_{k-j-i-1}$. Let $k^* = k-j-i-1$. Then the following information-forgetting state extension holds: $(k-j-i, \Psi'') \sqsubseteq (k^*, \Psi^*)$. Since $\langle k-j-i, \Psi'', v\rangle \in \tau_1$ and τ_1 is a type, we have $\langle k^*, \Psi^*, v\rangle \in \tau_1$. The definition of $\to$ then implies that $e[v/x] :_{k^*,\Psi^*} \tau_2$. But we now have $(S, \sigma(e_1\, e_2)) \longmapsto_M^{j+i+1} (S'', e[v/x])$, $(k, \Psi) \sqsubseteq (k^*, \Psi^*)$, $S'' :_{k^*} \Psi^*$, and $e[v/x] :_{k^*,\Psi^*} \tau_2$. By Definition 3.6 (Expr : Type), this means that if $(S'', e[v/x])$ generates a value in fewer than k^* steps then that value will be of type τ_2. Hence, we may conclude that $\sigma(e_1\, e_2) :_{k,\Psi} \tau_2$ as we wanted to show. $\square$

Theorem 3.21 (Application)
If Γ is a type environment, e_1 and e_2 are (possibly open) terms, and τ_1 and τ_2 are types such that $\Gamma \models_M e_1 : \tau_1 \to \tau_2$ and $\Gamma \models_M e_2 : \tau_1$ then $\Gamma \models_M (e_1\, e_2) : \tau_2$.

Proof: We must prove that under the premises of the theorem, for any $k \geq 0$, we have $\Gamma \models_M^k (e_1\, e_2) : \tau_2$. More specifically, for any σ and Ψ such that $\sigma :_{k,\Psi} \Gamma$ we must show $\sigma(e_1\, e_2) :_{k,\Psi} \tau_2$. From the premise $\Gamma \models_M e_1 : \tau_1 \to \tau_2$ we have $\sigma(e_1) :_{k,\Psi} \tau_1 \to \tau_2$. To show $\sigma(e_1\, e_2) :_{k,\Psi} \tau_2$, suppose $S :_k \Psi$ for some store S. Then, from $\sigma(e_1) :_{k,\Psi} \tau_1 \to \tau_2$ it follows that $(S, \sigma(e_1))$ is safe for k steps. Either $(S, \sigma(e_1))$ reduces for k steps without reaching a state (S', v_1) where v_1 is a value — in which case $(S, \sigma(e_1\, e_2))$ does not generate a value in less than k steps and hence $\sigma(e_1\, e_2) :_{k,\Psi} \tau_2$ (for any τ_2) — or the value v_1 is a lambda expression $\lambda x.e$. In the latter case, since $\sigma(e_1) :_{k,\Psi} \tau_1 \to \tau_2$ and $(S, \sigma(e_1)) \longmapsto_M^j (S', \lambda x.e)$, where $j < k$ and irred$(S', \lambda x.e)$, it follows that there exists a Ψ' such that $(k, \Psi) \sqsubseteq (k - j, \Psi')$ and

```
Theorem compat_app Γ e1 e2 A B :
  Γ ⊨ e1 : TArrow A B → Γ ⊨ e2 : A → Γ ⊨ App e1 e2 : B.
Proof.
  iIntros (e1Typed e2Typed ? ? ?) "#HΓ".
  use_bind (AppLCtx _) v1 "#Hv1" e1Typed.
  use_bind (AppRCtx _) v2 "#Hv2" e2Typed.
  by iApply "Hv1".
Qed.
```

Proof in Iris (in Coq)

$\langle k - j - i, \Psi'', v\rangle \in \tau_1$.

Pick memory typing $\Psi^* = \lfloor \Psi'' \rfloor_{k-j-i-1}$. Let $k^* = k - j - i - 1$. Then the following information-forgetting state extension holds: $(k - j - i, \Psi'') \sqsubseteq (k^*, \Psi^*)$. Since $\langle k - j - i, \Psi'', v\rangle \in \tau_1$ and τ_1 is a type, we have $\langle k^*, \Psi^*, v\rangle \in \tau_1$. The definition of $\to$ then implies that $e[v/x] :_{k^*,\Psi^*} \tau_2$. But we now have $(S, \sigma(e_1\, e_2)) \longmapsto_M^{j+i+1} (S'', e[v/x])$, $(k, \Psi) \sqsubseteq (k^*, \Psi^*)$, $S'' :_{k^*} \Psi^*$, and $e[v/x] :_{k^*,\Psi^*} \tau_2$. By Definition 3.6 (Expr : Type), this means that if $(S'', e[v/x])$ generates a value in fewer than k^* steps then that value will be of type τ_2. Hence, we may conclude that $\sigma(e_1\, e_2) :_{k,\Psi} \tau_2$ as we wanted to show. $\square$

Theorem 3.21 (Application).

$\sigma(e_1\,e_2) :_{k,\Psi} \tau_2$ (for any τ_2) — or the value v_1 is a lambda expression $\lambda x.e$. In the latter case, since $\sigma(e_1) :_{k,\Psi} \tau_1 \to \tau_2$ and $(S, \sigma(e_1)) \longmapsto^{j}_{M} (S', \lambda x.e)$, where $j < k$ and irred$(S', \lambda x.e)$, it follows that there exists a Ψ' such that $(k, \Psi) \sqsubseteq (k-j, \Psi')$ and

Iris dramatically simplifies the development of step-indexed models, while also being machine-checked!

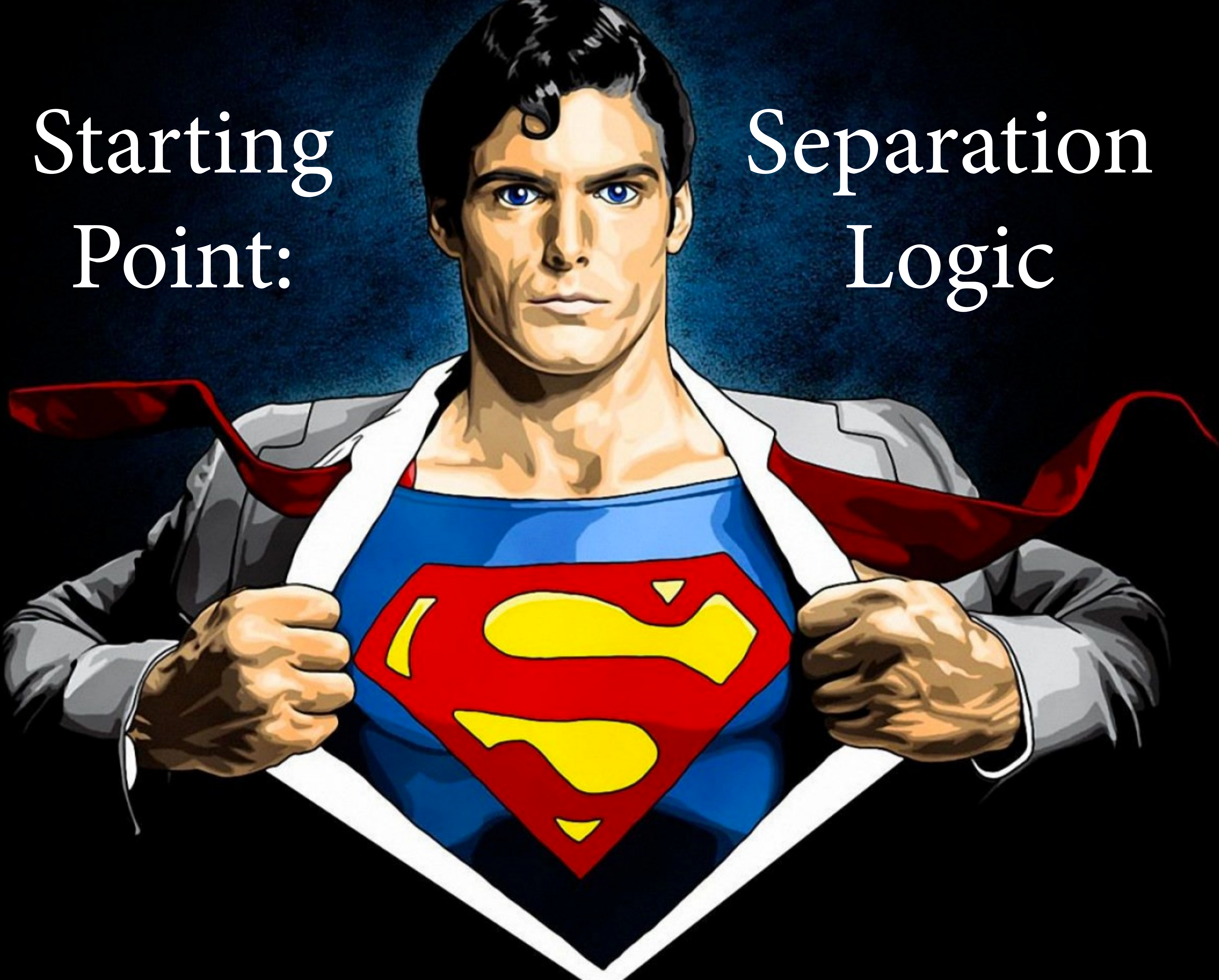
```
Theorem compat_app Γ e1 e2 A B :
  Γ ⊨ e1 : TArrow A B → Γ ⊨ e2 : A → Γ ⊨ App e1 e2 : B.
Proof.
  iIntros (e1Typed e2Typed ? ? ?) "#HΓ".
  use_bind (AppLCtx _) v1 "#Hv1" e1Typed.
  use_bind (AppRCtx _) v2 "#Hv2" e2Typed.
  by iApply "Hv1".
Qed.
```

Proof in Iris (in Coq)

$\langle k-j-i, \Psi'', v\rangle \in \tau_1$.

Pick memory typing $\Psi^* = \lfloor \Psi'' \rfloor_{k-j-i-1}$. Let $k^* = k-j-i-1$. Then the following information-forgetting state extension holds: $(k-j-i, \Psi'') \sqsubseteq (k^*, \Psi^*)$. Since $\langle k-j-i, \Psi'', v\rangle \in \tau_1$ and τ_1 is a type, we have $\langle k^*, \Psi^*, v\rangle \in \tau_1$. The definition of $\to$ then implies that $e[v/x] :_{k^*,\Psi^*} \tau_2$. But we now have $(S, \sigma(e_1\,e_2)) \longmapsto^{j+i+1}_{M} (S'', e[v/x])$, $(k, \Psi) \sqsubseteq (k^*, \Psi^*)$, $S'' :_{k^*} \Psi^*$, and $e[v/x] :_{k^*,\Psi^*} \tau_2$. By Definition 3.6 (Expr : Type), this means that if $(S'', e[v/x])$ generates a value in fewer than k^* steps then that value will be of type τ_2. Hence, we may conclude that $\sigma(e_1\,e_2) :_{k,\Psi} \tau_2$ as we wanted to show. □

Starting Point: Separation Logic



Starting Point: Separation Logic

Extension of Hoare logic (O'Hearn-Reynolds-…, 1999)

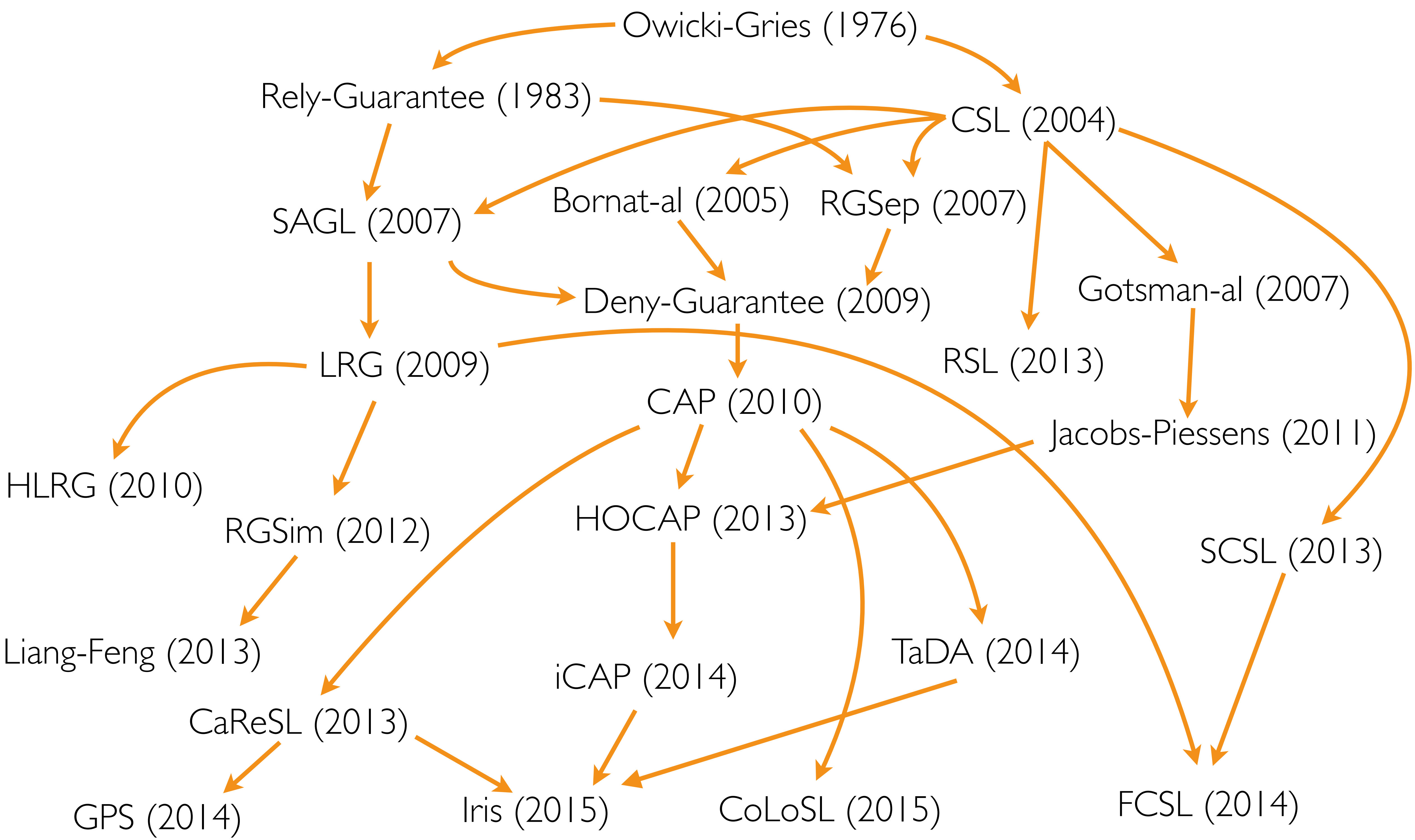
- For reasoning about pointer-manipulating programs

Major influence on many verification & analysis tools

- e.g. Infer, VeriFast, Viper, Bedrock, jStar, …

Separation logic = Ownership logic

- Perfect fit for modeling Rust's ownership types!



Owicki-Gries (1976)
Rely-Guarantee (1983)
CSL (2004)
SAGL (2007)
Bornat-al (2005)
RGSep (2007)
Deny-Guarantee (2009)
Gotsman-al (2007)
LRG (2009)
RSL (2013)
CAP (2010)
Jacobs-Piessens (2011)
HLRG (2010)
RGSim (2012)
HOCAP (2013)
SCSL (2013)
Liang-Feng (2013)
iCAP (2014)
TaDA (2014)
CaReSL (2013)
GPS (2014)
Iris (2015)
CoLoSL (2015)
FCSL (2014)

CaReSL:

$$\frac{\mathcal{C} \vdash \forall b \overset{\text{rely}}{\sqsupseteq_\pi} b_0.\ (\!|\pi[\![b]\!] * P|\!)\ i \Rrightarrow_{\mathrm{I}} a\ (\!|x.\ \exists b' \overset{\text{guar}}{\sqsupseteq_\pi} b.\ \pi[\![b']\!] * Q|\!)}{\mathcal{C} \vdash \left\{ \boxed{b_0}^{n}_{\pi} * \triangleright P \right\}\ i \Rrightarrow a\ \left\{x.\ \exists b'.\ \boxed{b'}^{n}_{\pi} * Q\right\}}\ \textsc{UpdIsl}$$

iCAP:

$$\frac{\begin{array}{c}\Gamma, \Delta \mid \Phi \vdash \mathsf{stable}(\mathsf{P}) \qquad \Gamma, \Delta \mid \Phi \vdash \forall y.\ \mathsf{stable}(\mathsf{Q}(y)) \\ \Gamma, \Delta \mid \Phi \vdash n \in C \qquad \Gamma, \Delta \mid \Phi \vdash \forall x \in X.\ (x, f(x)) \in \overline{T(A)} \vee f(x) = x \\ \Gamma \mid \Phi \vdash \forall x \in X.\ (\Delta).\langle \mathsf{P} * \circledast_{\alpha \in A}[\alpha]^{\mathsf{n}}_{g(\alpha)} * \triangleright I(x) \rangle\ c\ \langle \mathsf{Q}(x) * \triangleright I(f(x)) \rangle^{C \setminus \{n\}}\end{array}}{\begin{array}{c}\Gamma \mid \Phi \vdash (\Delta).\ \langle \mathsf{P} * \circledast_{\alpha \in A}[\alpha]^{\mathsf{n}}_{g(\alpha)} * \mathsf{region}(X, T, I, n) \rangle \\ c \\ \langle \exists x.\ \mathsf{Q}(x) * \mathsf{region}(\{f(x)\}, T, I, n) \rangle^{C}\end{array}}\ \textsc{Atomic}$$

TaDA:

Use atomic rule

$$\frac{\begin{array}{c} a \notin \mathcal{A} \quad \forall x \in X.\ (x, f(x)) \in \mathcal{T}_{\mathbf{t}}(\mathrm{G})^{*} \\ \lambda; \mathcal{A} \vdash \mathbb{W} x \in X.\ \left\langle p_p \,\middle|\, I(\mathbf{t}^{\lambda}_{a}(x)) * p(x) * [\mathrm{G}]_a \right\rangle\ \mathbb{C}\ \ \exists\!\!\exists y \in Y.\ \left\langle q_p(x, y) \,\middle|\, I(\mathbf{t}^{\lambda}_{a}(f(x))) * q(x, y) \right\rangle \end{array}}{\lambda + 1; \mathcal{A} \vdash \mathbb{W} x \in X.\ \left\langle p_p \,\middle|\, \mathbf{t}^{\lambda}_{a}(x) * p(x) * [\mathrm{G}]_a \right\rangle\ \mathbb{C}\ \ \exists\!\!\exists y \in Y.\ \left\langle q_p(x, y) \,\middle|\, \mathbf{t}^{\lambda}_{a}(f(x)) * q(x, y) \right\rangle}$$

2007)

(2011)

HLRG (20

L (2013)

Liang-Feng

CaReSL (2013)

GPS (2014)

Iris (2015)

CoLoSL (2015)

FCSL (2014)

CaReSL: $$\frac{\mathcal{C} \vdash \forall b \overset{\text{rely}}{\sqsupseteq_\pi} b_0.\ (\!|\pi[\![b]\!] * P|\!)\ i \Rightarrow_{\mathrm{I}} a\ (\!|x.\ \exists b' \overset{\text{guar}}{\sqsupseteq_\pi} b.\ \pi[\![b']\!] * Q|\!)}{\mathcal{C} \vdash \left\{\boxed{b_0}^{n}_{\pi} * \triangleright P\right\}\ i \Rightarrow a\ \left\{x.\ \exists b'.\ \boxed{b'}^{n}_{\pi} * Q\right\}}\ \textsc{UpdIsl}$$

iCAP: $\Gamma, \Delta \mid \Phi \vdash \mathsf{stable(P)}$

$\Gamma, \Delta \mid \Phi \vdash n \in C$

$\Gamma \mid \Phi \vdash \forall$ ...

No way to compose proofs from different separation logics!

TaDA: Use atomic rule

$$\frac{a \notin \mathcal{A} \quad \forall x \in X.\, (x, f(x)) \in \mathcal{T}_{\mathbf{t}}(\mathrm{G})^* \qquad \lambda; \mathcal{A} \vdash \mathbb{V} x \in X.\, \langle p_p \mid I(\mathbf{t}^{\lambda}_{a}(x)) * p(x) * [\mathrm{G}]_a \rangle\ \mathbb{C}\ \exists\!\!\exists y \in Y.\, \langle q_p(x,y) \mid I(\mathbf{t}^{\lambda}_{a}(f(x))) * q(x,y) \rangle}{\lambda + 1; \mathcal{A} \vdash \mathbb{V} x \in X.\, \langle p_p \mid \mathbf{t}^{\lambda}_{a}(x) * p(x) * [\mathrm{G}]_a \rangle\ \mathbb{C}\ \exists\!\!\exists y \in Y.\, \langle q_p(x,y) \mid \mathbf{t}^{\lambda}_{a}(f(x)) * q(x,y) \rangle}$$

HLRG (20

Liang-Fen

CaReSL (2013)

GPS (2014)

Iris (2015)

CoLoSL (2015)

FCSL (2014)

2007)

(2011)

(2013)

Key Idea of Iris

- Unify the field of separation logic using a single powerful mechanism:

Higher-order **ghost state**

Enables encoding of **step-indexed** models

Enables users to define custom **resources**

(see "Iris from the Ground Up", JFP'18, for details)

Key Idea of Iris

- Unify the field of separation logic using a single powerful mechanism:

With higher-order ghost state,
Iris lets you **derive** and **compose** advanced proof rules within one unifying framework.

step-indexed models custom **resources**

(see "Iris from the Ground Up", JFP'18, for details)

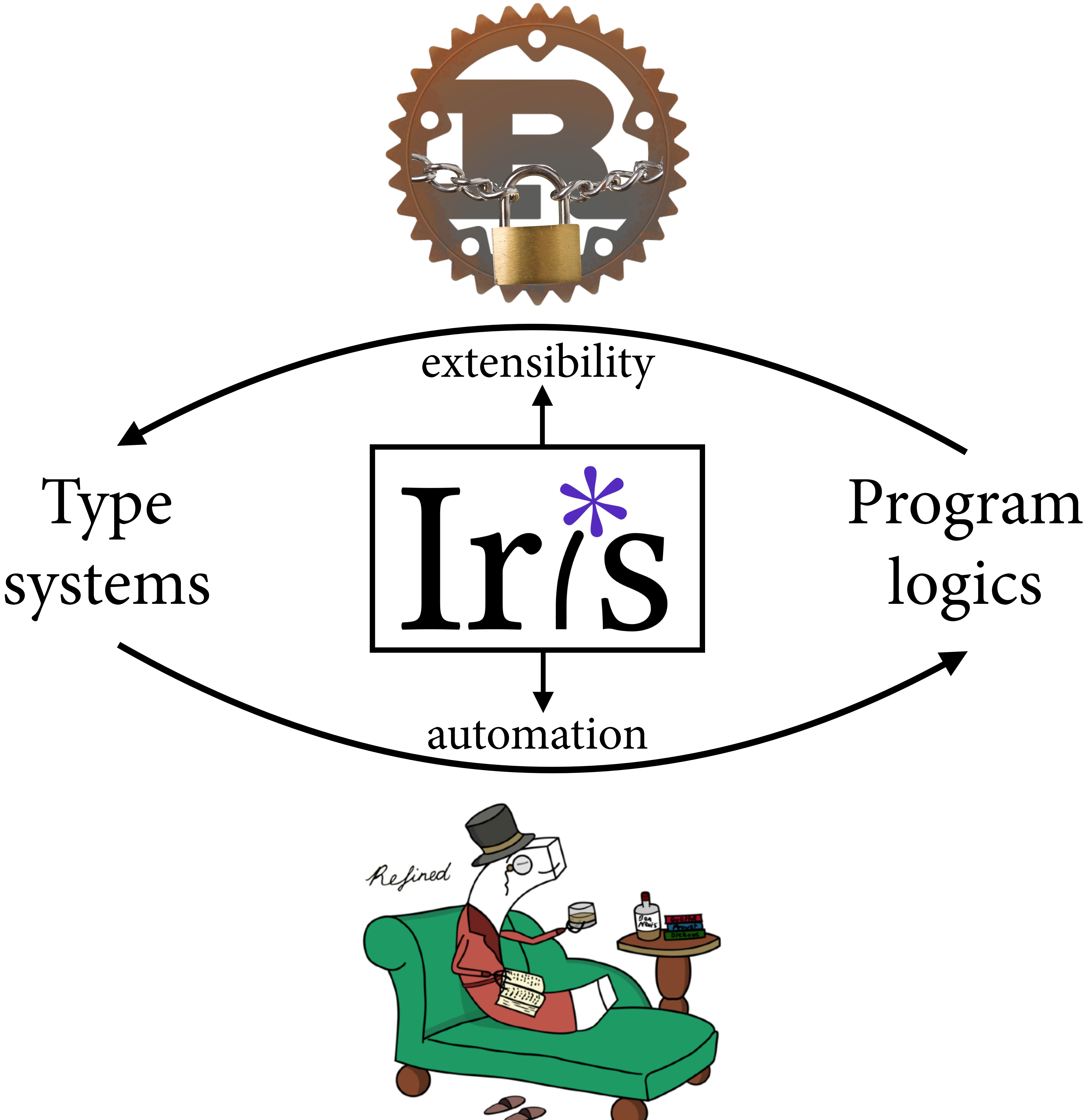
Impact of Iris & RustBelt

Iris

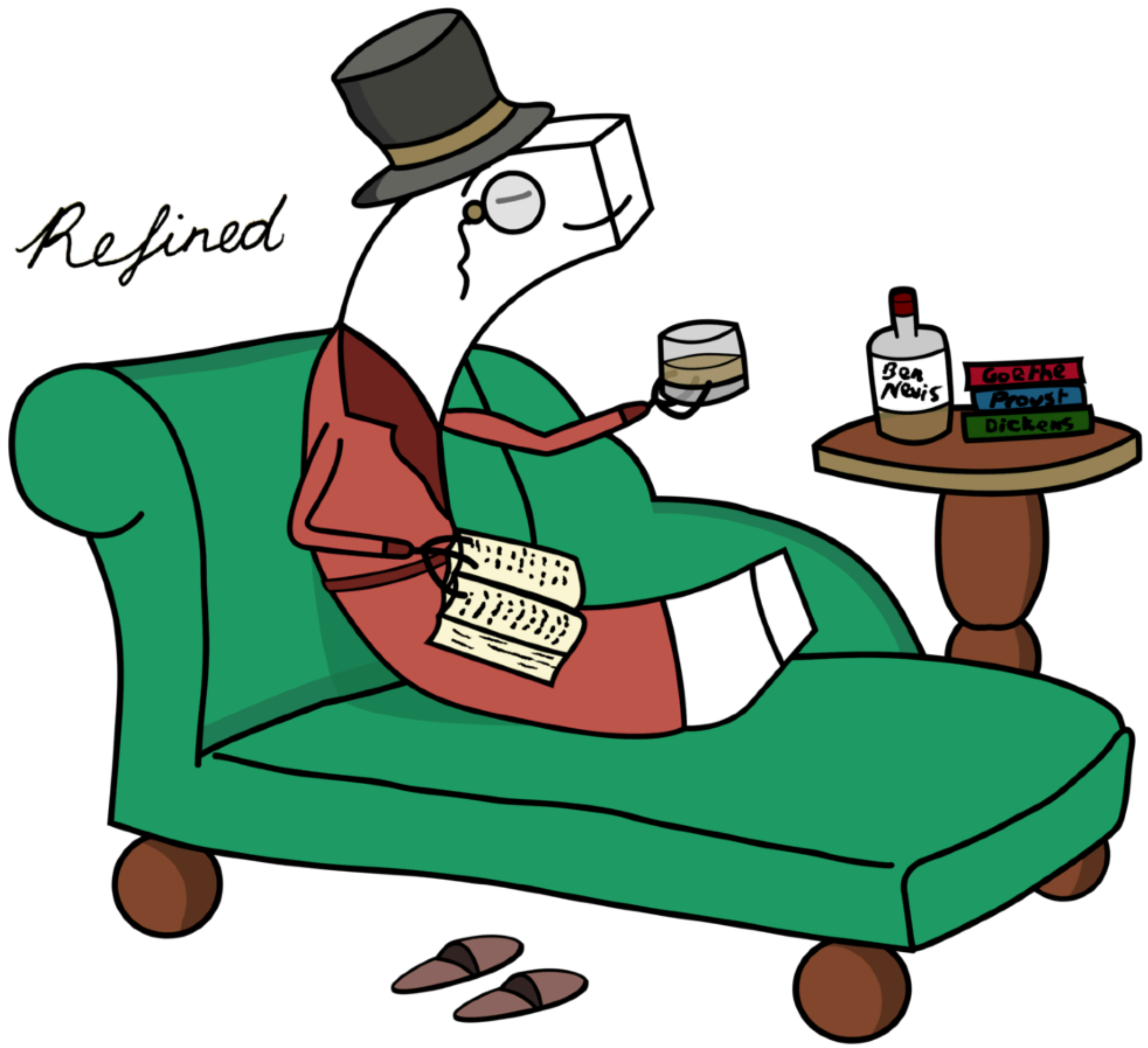
- 60 papers (28 in POPL/PLDI), 7 PhD theses
- Adopted as core tech. by systems verification researchers at MIT, BedRock, Meta

RustBelt

- Most-cited POPL/PLDI paper of 2018
- Pioneering effort in Rust verification & using program logics to prove extensible type safety


extensibility
Type systems
Iris
Program logics
automation
Refined

Refined



RefinedC

First verification tool for C programs that is

- **Automated**: user gives only specs/annotations
- **Foundational**: generates proofs in Coq

How?

- **Refinement type system** to encode functional invariants on C data types
- **Semantic model** of RefinedC types in Iris
- RefinedC typing rules formulated in **Lithium**, a restricted, automatable fragment of Iris

C:

```c
void append(list_t *l, list_t k) {
  if(*l == NULL) {
    *l = k;
  } else {
    append(&(*l)->next, k);
  }
}
```

Iris:

```
iIntros (p) "[Hxs Hys] H".
iLob as "IH" forall (l xs l' ys p).
destruct xs as [| x xs']; iSimplifyEq.
- wp_rec. wp_let. wp_match. by iApply "H".
- iDestruct "Hxs" as (l0 hd0) "(% & Hx & Hxs)".
  iSimplifyEq. wp_rec. wp_let. wp_match. wp_load.
  wp_let. wp_proj. wp_bind (app _ _)%E.
  iApply ("IH" with "Hxs Hys"). iNext. iIntros.
  wp_let. wp_proj. wp_store. iSimplifyEq. iApply "H".
  iExists l0, v. iFrame. done.
```

C:

```c
void append(list_t *l, list_t k) {
  if(*l == NULL) {
    *l = k;
  } else {
    append(&(*l)->next, k);
  }
}
```

RefinedC:

```
repeat liRStep; liShow.
```

C:

```c
void append(list_t *l, list_t k) {
  if(*l == NULL) {
    *l = k;
  } else {
    append(&(*l)->next, k);
  }
}
```

Distinguished paper and artifact awards at PLDI'21

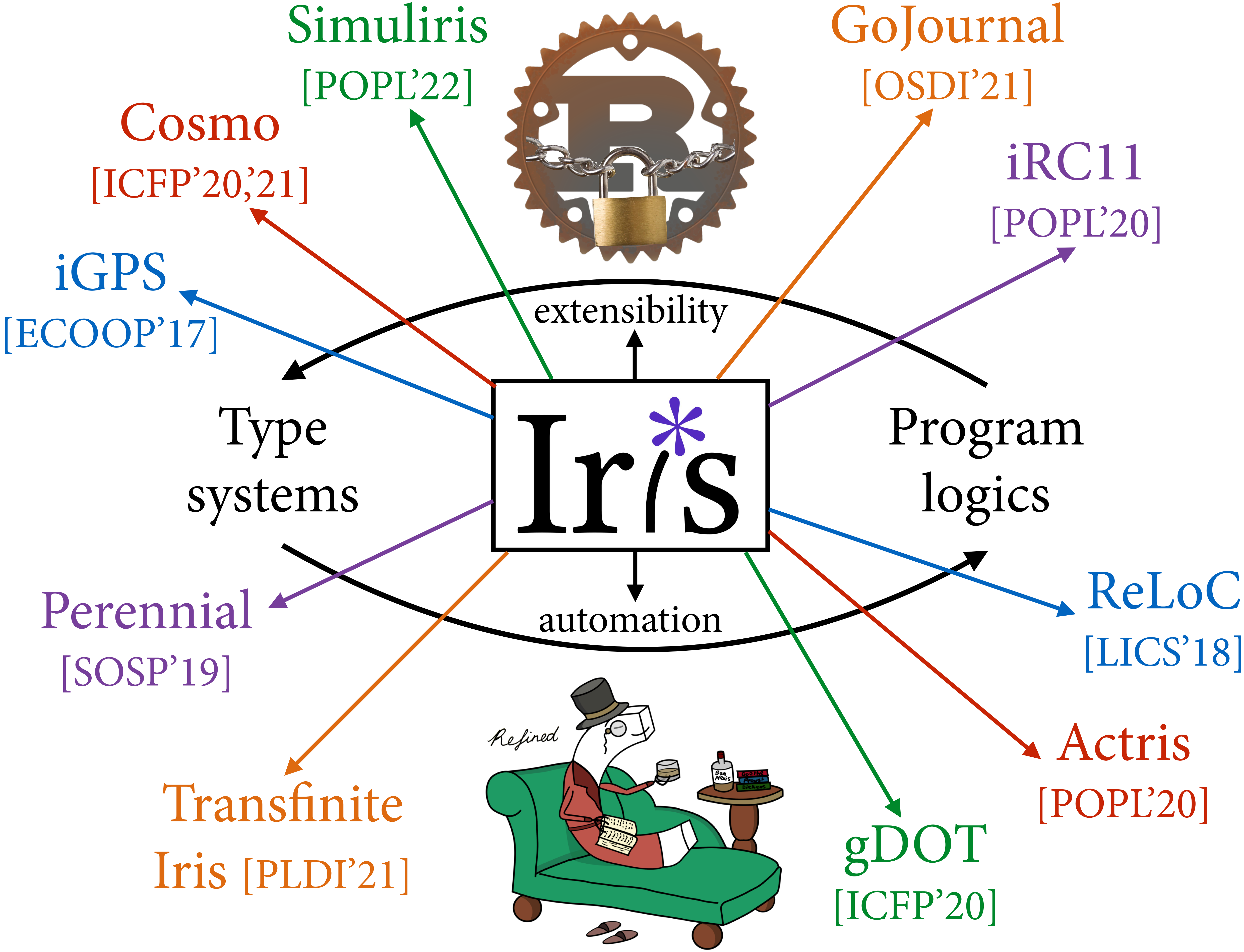
extensibility

Type systems

Iris

Program logics

automation


Simuliris
[POPL'22]
GoJournal
[OSDI'21]
Cosmo
[ICFP'20,'21]
iRC11
[POPL'20]
iGPS
[ECOOP'17]
extensibility
Type
systems
Iris
Program
logics
automation
Perennial
[SOSP'19]
ReLoC
[LICS'18]
Refined
Actris
[POPL'20]
Transfinite
Iris [PLDI'21]
gDOT
[ICFP'20]

Research Vision

Integrating Verification into Real-World Systems

Systems Verification

Impressive sys. verif. projects in past 15 years:

seL4 Security. Performance. Proof.

CertiKOS

deep spec

COMPCERT

Systems Verification

Impressive sys. verif. projects in past 15 years:

seL4 Security. Performance. Proof.   CERTIKOS   deep spec   COMPCERT

Some key **limitations** to their scope:

- Centered exclusively around the C language
- Idealized code/semantics to simplify verif.
- Huge manual proof effort by experts

Systems Verification

Impressive sys. verif. projects in past 15 years:

Goal: Develop systems verification tools that overcome these limitations!

- Centered exclusively around the C language
- Idealized code/semantics to simplify verif.
- Huge manual proof effort by experts

Direction #1: Rust Verification

✓ Safe + **unsafe** code

✗ Verif. is **manual**

P∗rust→∗i

(from ETH)

✓ Verif. is **automated**

✗ Only **safe** code

Direction #1: Rust Verification

✓ Safe + **unsafe** code

✗ Verif. is **manual**

P∗rust→∗i
(from ETH)

✓ Verif. is **automated**

✗ Only **safe** code

Goal: Tool that's automated & handles unsafe code

- In development: RustHornBelt, RefinedRust
- Possible verification goal: Redox microkernel

Direction #2: Realism

Prior work employs idealized coding/semantics

- Restricts coding patterns (e.g. allocate all data in one big array, prohibit taking address of local variables)
- Assumes strong concurrency semantics (e.g. SC)
- Uses idealized model of low-level system features (e.g. virtual memory, interrupts, exceptions, TLB)

Direction #2: Realism

Prior work employs idealized coding/semantics

- Restricts coding patterns (e.g. allocate all data in one big array, prohibit taking address of local variables)
- Assumes strong concurrency semantics (e.g. SC)
- Uses idealized model of low-level system features (e.g. virtual memory, interrupts, exceptions, TLB)

Ongoing: Verifying Linux pKVM hypervisor

- To be deployed on billions of Android devices
- Goal: Use RefinedC to verify Armv8 machine code against <u>authoritative</u> Armv8 semantics
- Joint with Sewell, Hur, et al., funded by Google

Direction #3: Usability

Problem: Even automated tools like RefinedC involve some annotation burden

- e.g. users must write tricky specs **manually**

```
struct [[rc::refined_by("a: nat")]] mem_t {
  [[rc::field("a @ int<size_t>")]] size_t len;
  [[rc::field("&own<uninit<a>>")]] unsigned char* buffer;
};

[[rc::parameters("a: nat", "n: nat", "p: loc")]]
[[rc::args    ("p @ &own<a @ mem_t>", "n @ int<size_t>")]]
[[rc::returns("{n≤a} @ optional<&own<uninit<n>>, null>")]]
[[rc::ensures("own p : {n ≤ a ? a - n : a} @ mem_t")]]
void* alloc(struct mem_t* d, size_t sz) {
  if(sz > d->len) return NULL;
  d->len -= sz;
  return d->buffer + d->len;
}
```

Direction #3: Usability

Problem: Even automated tools like RefinedC involve some annotation burden

- e.g. users must write tricky specs **manually**

```
struct [[rc::refined_by("a: nat")]] mem_t {
  [[rc::field("a @ int<size_t>")]] size_t len;
  [[rc::field("&own<uninit<a>>")]] unsigned char* buffer;
};

[[rc::parameters("a: nat", "n: nat", "p: loc")]]
[[rc::args   ("p @ &own<a @ mem_t>", "n @ int<size_t>")]]
[[rc::returns("{n≤a} @ optional<&own<uninit<n>>, null>")]]
[[rc::ensures("own p : {n ≤ a ? a - n : a} @ mem_t")]]
void* alloc(struct mem_t* d, size_t sz) {
  if(sz > d->len) return NULL;
  d->len -= sz;
  return d->buffer + d->len;
}
```

Idea: Infer specs using **biabduction** (POPL'09)

- Inferring full functional correctness specs is likely impractical, but precision can be improved by allowing user to “sketch” specs

Thank you!

`iris-project.org`