Neurosymbolic C++ Verification using Rocq

Lennart Beringer Jasper Haag Rodolphe Lepigre **Gregory Malecha** Rayan Soban Ehtesham Zahoor

SkyLabs AI

What if you had to verify this?

```
/*
 * Lookup OBJ capability for the specified selector
 *
 * @param sel   Selector whose capability is being looked up
 * @return      Object Capability (if slot is non-empty) or Null Capability (otherwise)
 */
Capability Space_obj::lookup (unsigned long sel) const
{
    auto l { lev }; uintptr_t cte;

    // Walk down the capability tables from the root, computing the slot index at each level
    for (auto ptr { &root };; ptr = &table (cte)->slot[(sel >> --l * bpl) % Captable::entries]) {

        // Return capability upon reaching the last existing or leaf level
        if (!(cte = *ptr) || !l)
            return Capability { cte };
    }
}
```

```
static constexpr auto entries { BIT (bpl) };

using entry_t = Atomic<uintptr_t, __ATOMIC_ACQUIRE>;
entry_t slot[entries] {};
```

- C++
- Low-level code
- Fine-grained concurrency
- Hardware interaction

NOVA Microhypervisor: https://gitlab.com/bluerocksec/NOVA

What if you had to verify this?

- C++
- Low-level code
- Fine-grained concurrency
- Hardware interaction
- **Strong specifications**

```
/*
 * Lookup OBJ capability for the specified selector
 *
```

```
(** The continuation [K] lets the caller relate the capability
    returned by [lookup] to the capability [cap] known (by the
    caller's proof) to be bound to [sel]. *)
Definition do_lookup' (E : coPset) (g : spc.obj.gname) (sel : sel.t)
    (K : capability.t → mpred) : mpred :=
  AR1 << ∃∃ q cap, space_obj.entry g sel q cap >>  (** current entry for [sel] *)
      @ top \ E, empty
      << K cap >>.
Definition do_lookup := do_lookup' (↑tableN).
cpp.spec "Space_obj::lookup(unsigned long) const" as lookup from (module) with
  \this this
  \spec@{WpSpec_cpp}
  \arg{sel} "sel" (sel.V sel)                (** selector to look up *)
  \pre{g}    this |-> space_obj.I g          (** space invariant *)
  \prepost{q pd} this |-> space_obj.R g q pd  (** space representation *)
  \prepost         const.all
  \pre{K}     do_lookup g sel K
  \post{v (cap : capability.t)}[Vref v]
    [| published cap |] **
    K cap **                                 (** found capability [cap] *)
    v |-> capability.R 1$m cap          (** and returned it in value [v] *)
).
```

NOVA Microhypervisor: https://gitlab.com/bluerocksec/NOVA

What if you had to verify this?

- C++
- Low-level code
- Fine-grained concurrency
- Hardware interaction
- Strong specifications

```
/*
 * Lookup OBJ capability for the specified selector
 *
```

```
(**

Defi

  AR

Defi
cpp.
  \t
  \s
  \a
  \p
  \p
  \p
  \p
  \p

).
```

```
Lemma lookup_ok : verify[module] "Space_obj::lookup(unsigned long) const".
Proof using All.
  verify_spec.
  go.

  wp_for (λ ρ ⇒
    Exists p (n : Z),
    _local ρ "ptr" |-> ptrR<entryT> 1$m p **
    p |-> type_ptrR entryT **
    _local ρ "l" |-> intR 1$m n **
    [| (0 ≤ n ≤ sel.lev)%Z |] **
    _local ρ "cte" |-> anyR Tuintptr_t 1$m **
    atomic.do_load (t:=Tuintptr_t) (temp_args:=atomic.default_args)
      p (do_load_K cns g q sel K n (base_idx n sel))
  ).

  go.
  rewrite {1}/do_load_K /read_case.
  vc_split.
  { (* return with nullptr *) go. }

  go.
  rewrite /captableNonNull /peeked_slot_type_ptr.
  wp_if.
  { (* return with l = 0 *) go. }
  (* continue to loop *)
  go.
Qed.
```

NOVA Microhypervisor: https://gitlab.com/bluero

What if yo
verify this?

- C++
- Low-level coc
- Fine-grained
- Hardware inte
- Strong specifi

NOVA Microhypervisor:

BlueRock / Formal Methods

Home Team Blogs Tech Reports Talks FAQs

https://bluerocksec.gitlab.io/formal-methods/blogs/2026-03-31-NOVA-relaxed-memory/

NOVA is getting relaxed-memory ready (26.08.0)

March 31, 2026

Tags: NOVA-proofs, NOVA-status

From the February 2026 release (26.08.0), the implementation of NOVA has been fixed with more correct uses of relaxed memory accesses. More fixes and documentation will come in the next releases. Accordingly, the formal spec (`nova_interface`) has been updated to expose relaxed memory effects that cannot be hidden to client code. The formal proof (`nova_proof`) has also been updated to the new spec, with several remaining TODOs concerning the incomplete specification of RCU (garbage collection).

Release-Acquire is now the default in the implementation

Previous releases of NOVA have been using either relaxed accesses (`memory_order_relaxed`) or sequentially consistent accesses (SC, `memory_order_seq_cst`), which are either too weak or too strong in their use cases. From the February 2026 release, NOVA has switched to use release-acquire (`memory_order_acquire`, `memory_order_release`, `memory_order_acq_rel`) as the default.

NOVA now uses relaxed accesses in case no synchronization is required, such as updates to an `Ec`'s continuation field when in sequential mode, or external synchronization is required from clients, such as in `ctrl_pt`. Meanwhile, NOVA uses sequentially consistent accesses to forbid undesirable reordering behaviors, such as when the per-CPU `current` Ec field is updated (context-switching) concurrently with other CPUs reading that field to check if a TLB invalidation or a recall is needed.

The formal spec exposes relaxed behaviors

The uses of weaker-than-SC accesses now expose weaker behaviors to NOVA clients. For example, if clients now use `ctrl_pt` concurrently, stale values of a `Pt`'s PID or MTD can be observed, and clients need to provide sufficient external synchronization to observe the latest values. The updated nova_spec for `ctrl_pt` now exposes a history of readable values for both PID and MTD. From this spec, clients (such as the BlueRock virtualization user code) can derive a sequential spec where the latest value will be observed.

The formal proof uses iRC11 relaxed memory logic

To verify code using relaxed memory, the `nova_proof` axiomatizes the rules of the iRC11 concurrent separation logic for relaxed

Just the beginning…

Many examples in this talk are deliberately small *for illustration purposes*.

However, these tools are built for working with production code.

- NOVA Microhypervisor: https://gitlab.com/blueroсksec/NOVA
- Blockchain: https://github.com/category-labs/monad
- Device Drivers: https://sel4.systems/Summit/2025/slides/verified-zynqmp.pdf
- C++ Standard Library: https://github.com/skylabsai/brick-libcpp/

* https://bluerocksec.gitlab.io/formal-methods/tech_reports/

A Formal Specification of the NOVA Microhypervisor's ABI *

Verifying a Virtual Machine Monitor *

Modular, Full-System Verification *

Gregory Malecha
BlueRock Security, Inc
USA
gregory@bluerock.io

Hoang-Hai Dang
BlueRock Security, Inc
Germany
hai@bluerock.io

Paolo G. Giarrusso
BlueRock Security, Inc
Germany
paolo@bluerock.io

Simon Hudon
BlueRock Security, Inc
USA
simon@bluerock.io

Jan-Oliver Kaiser
BlueRock Security, Inc
Germany
janno@bluerock.io

David Swasey*
Riverside Research, Inc
USA
pswasey@riversideresearch.org

ABSTRACT

We present an approach to specifying operating systems that is both highly modular and supports deriving various properties necessary to modern operating systems. The approach combines a machine semantics decomposed by privilege levels with a separation logic specification of the operating system API. We describe how the specification style enables natural proofs of robust safety in addition to deep behavioral refinements of user-mode applications running atop the OS. This approach enables simple and flexible concurrent specifications and unlocks new opportunities for whole-system verification.

KEYWORDS

Operating system specification, Formal verification, Multi-core verification, Separation logic, Iris

ACM Reference Format:
Gregory Malecha, Hoang-Hai Dang, Paolo G. Giarrusso, Simon Hudon, Jan-Oliver Kaiser, and David Swasey. 2025. Modular, Full-System Verification. In *Workshop in Hot Topics in Operating Systems (HOTOS 25), May 14–16, 2025, Banff, AB, Canada.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3713082.3730387

1 INTRODUCTION

Systems-software verification has been an active area of research for almost two decades. Existing work lays a foundation pieces of software. The verifications are built around monolithic (whole-program) proof principles [11] or rely on highly stylized coding patterns [8]. They are also limited in their ability to capture concurrency and rely on either completely eliminating it [11] or require all specifications to be linearizable [8]. While these approaches simplify reasoning, they do not account for the complexities of modern systems.

We aim for an operating system *specification* that is

(1) **Realistic**. Modern OS APIs are rarely atomic, and concurrently executing system calls can lead to subtle behaviors that the specification must account for. These include weak memory behaviors that are pervasive on modern platforms [18, 21, 20].
(2) **Two-sided** [1]. It must be usable not only to verify the operating system itself, but also to support user-mode reasoning on top of the OS. In particular, clients should be able to prove strong functional correctness properties of user-mode code, including behavioral refinements, using only the OS specification.
(3) **Modular**. It must scale to support the large and complex APIs present in modern systems. This demands both thread- and component-local specifications that are capable of encapsulating implementation details.
(4) **Robustly Safe** [7]. It must support running *arbitrary, untrusted* user-mode code and enable user-mode applications to use the OS specification to establish their

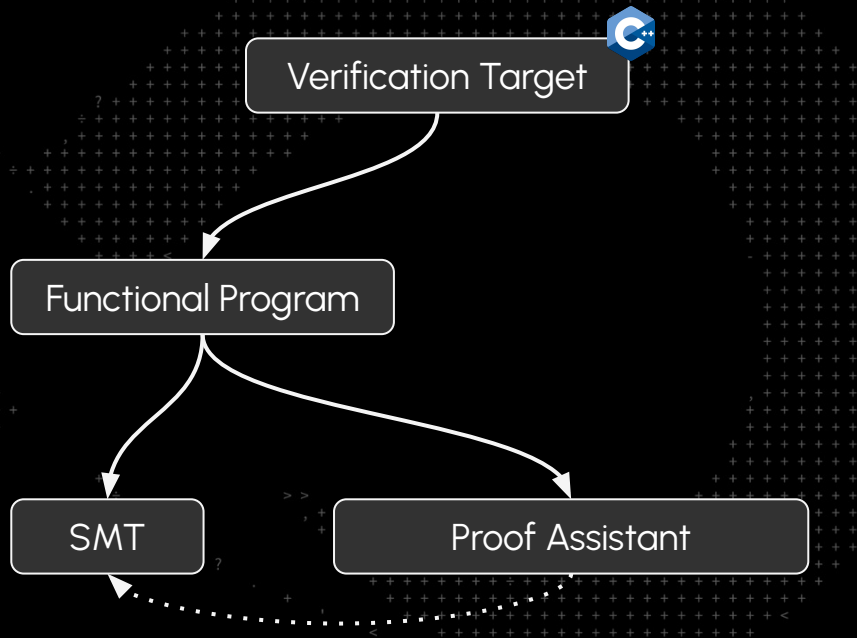
Program Verification Strategies

Most verification methodologies go through functional programs.

- Clean semantics
- Great tools for automatic and auto-active verification, e.g. SMT, Dafny, etc.

Functional programs are not an ideal verification IR.

- Translation strategy is often fixed ahead of time.
- Concurrency is difficult to model and reason about.



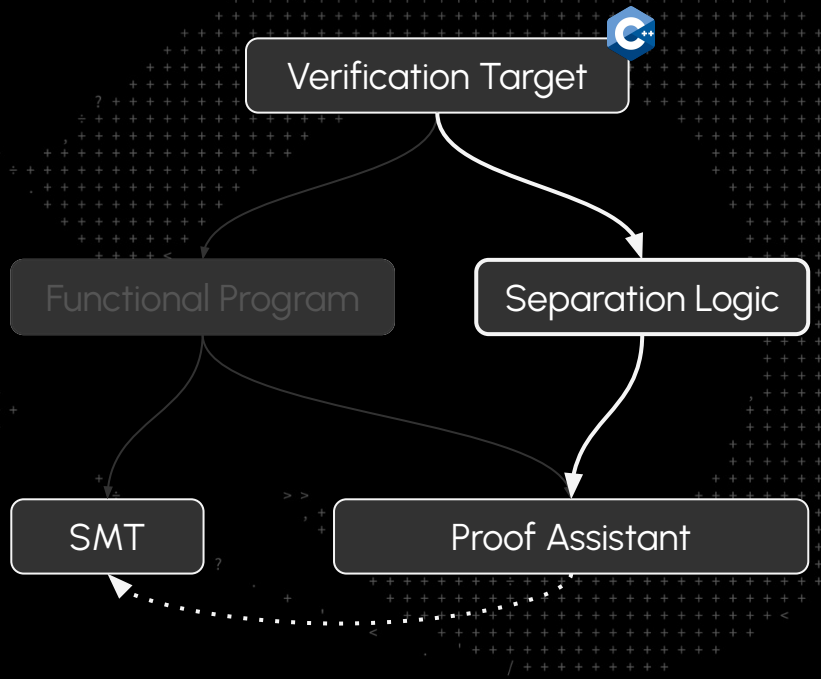
Program Verification Strategies

Separation logic is the logic independence.

- A (universal) **ownership discipline** on program state.
- Dynamic semantics by consuming & producing resources.
- **Language-agnostic** formalism

Tools built on separation logic:

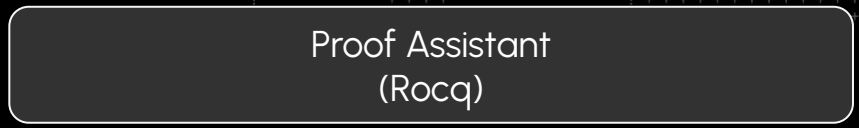
- VST, Viper, RefinedC, RefinedRust, etc.
- BRiCk – more on this later



A Foundation for Software Verification

Proof assistants provide

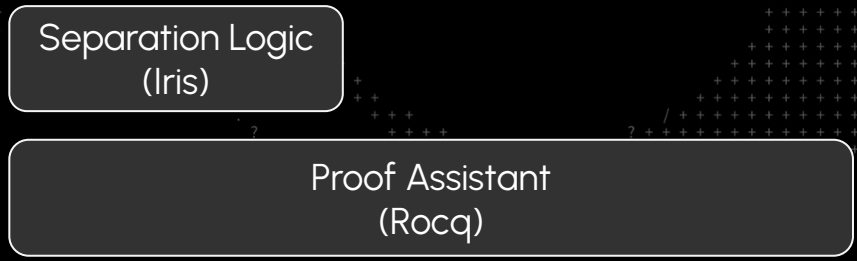
- Highly expressive language for specifications and proofs
- A way to **integrate evidence** with a small TCB
- Domain-relevant theories, e.g. floating point, labeled transition systems, etc.



A Foundation for Software Verification

Iris is the state-of-the-art in separation logic

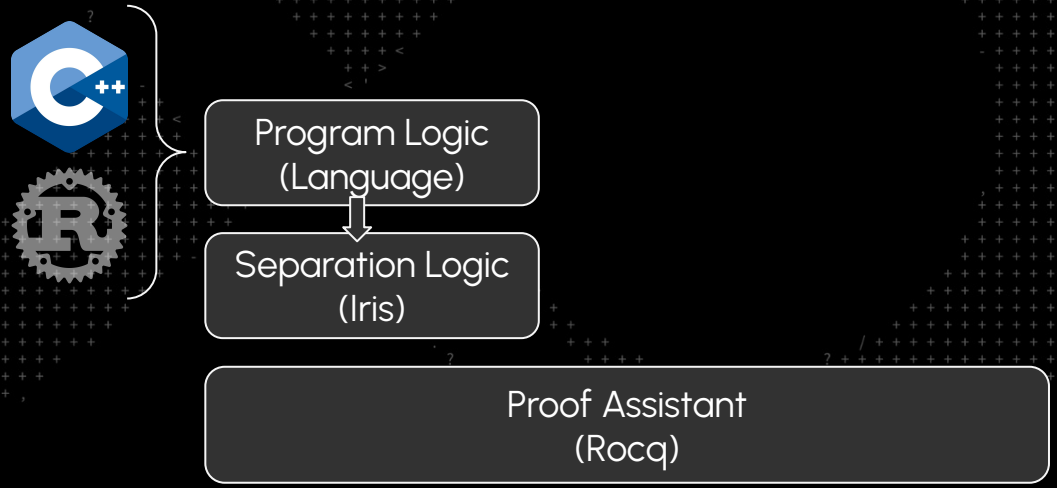
- Foundational library for separation logic
- Step-indexed logic for impredicative reasoning
- Powerful support for custom CMRAs
- Proof mode (IPM) for doing "manual" proofs



A Foundation for Software Verification

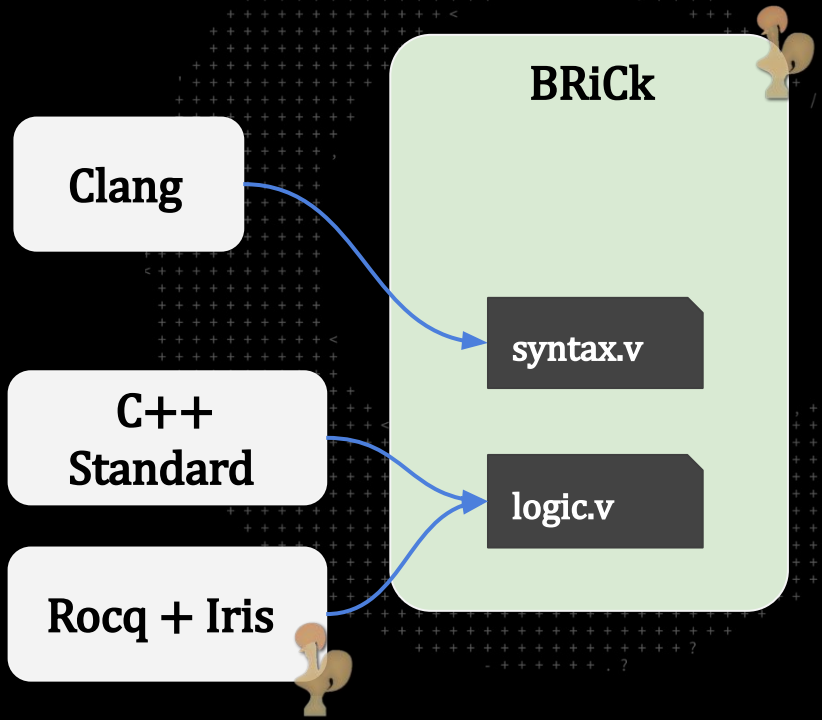
Define a logic for the programming language.

- A (universal) **ownership discipline** on program state.
- **Predicate transformers** to express the behavior of **language constructs**.
- Highly modular. Can easily support language subsets.



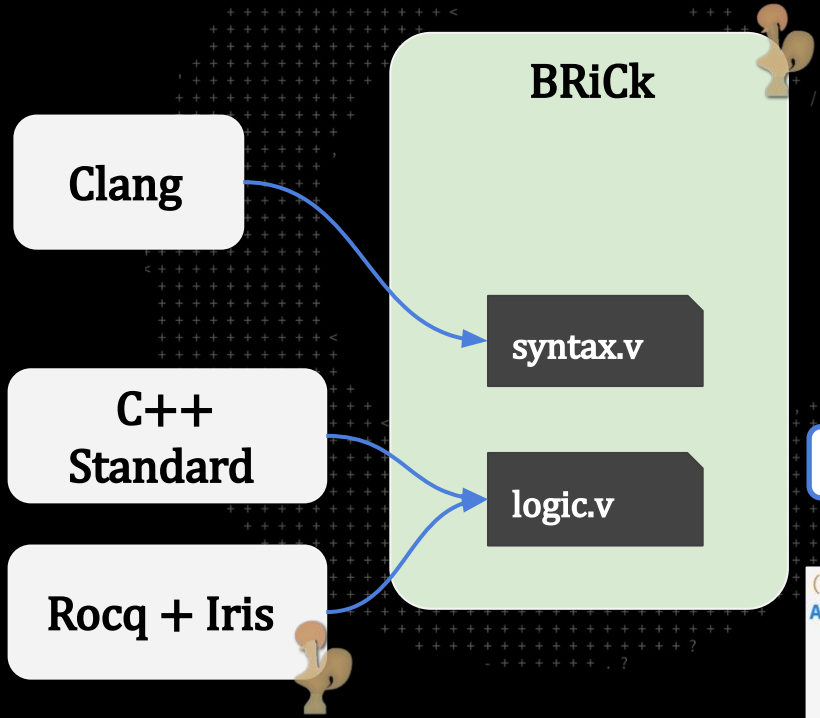
BRiCk C++ Infrastructure

https://github.com/skylabsai/brick



BRiCk C++ Infrastruct

https://github.com/skylabsai/brick




```
(** [Cl2r] represents reads of locations.
This counts as an _access_, so it must happen at one of the types listed in
https://eel.is/c++draft/basic.lval#11.
*)
Axiom wp_operand_cast_l2r : ∀ e Q,
  (letI* a, free := wp_glval e in
   ∃ v,
     (∃ q, a |-> tptsto_fuzzyR (erase_qualifiers $ type_of e) q v ** T) ∧
     Q v free)
  ⊢ wp_operand (Ecast Cl2r e) Q.
```

Program state reflected in separation logic.

```
(** The `*` operator is an lvalue, but this does not perform an
    access (see [wp_operand_cast_l2r] instead).

    We check pointer [p] is strictly valid and aligned.
    The standard says (<https://eel.is/c++draft/expr.unary.op#1>):

    > The unary * operator performs indirection: the expression
    > to which it is applied shall be a pointer to an object type,
    > or a pointer to a function type and the result is an lvalue
    > referring to the object or function to which the expression
    > points. If the type of the expression is “pointer to T”, the
    > type of the result is “T”.
 *)
Axiom wp_lval_deref : ∀ ty e Q,
    letI* v , free := wp_operand e in
    ∃ p, [| v = Vptr p |] **
    reference_to (erase_qualifiers ty) p **
    Q p free
    ⊢ wp_lval (Ederef e ty) Q.
```

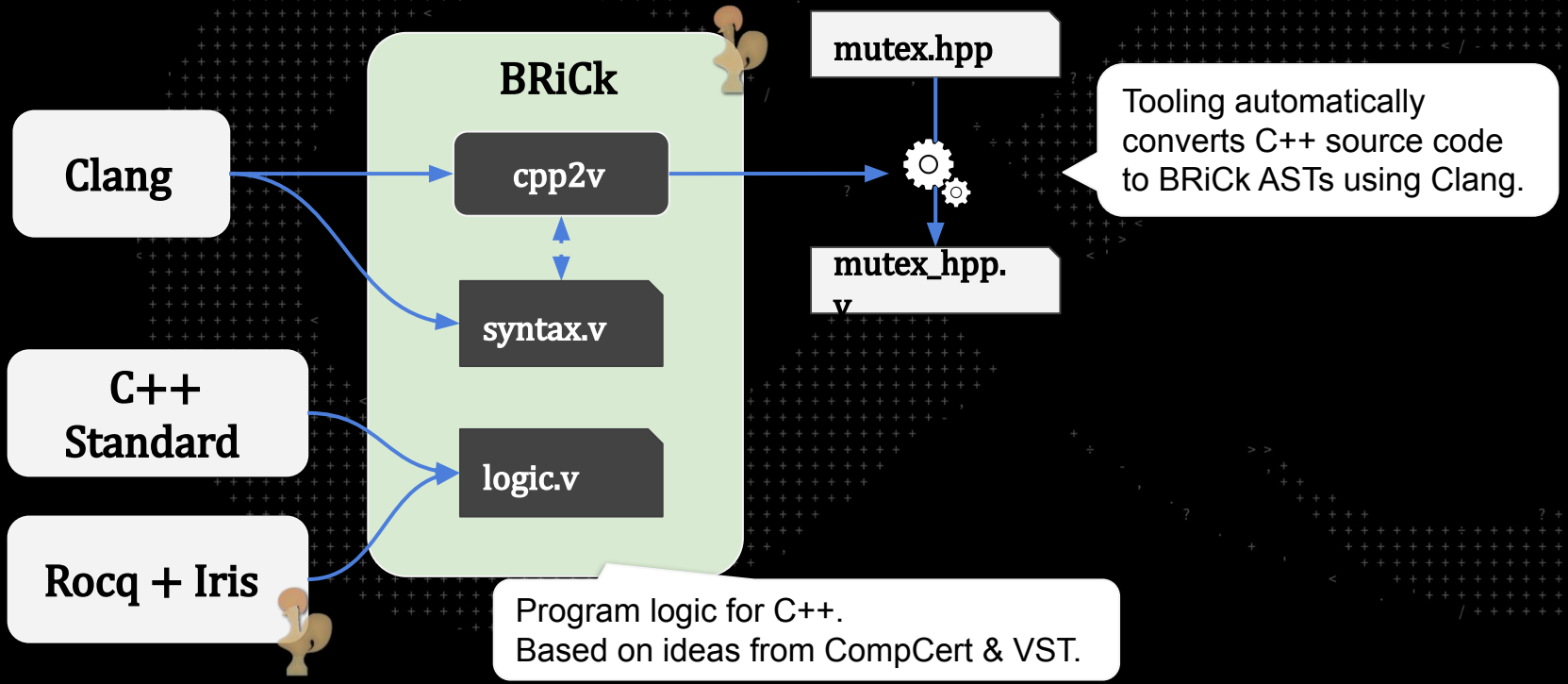
Object lifetime and bounds checking

```
(** Evaluation of binary operators. *)
Axiom wp_operand_binop : ∀ o e1 e2 ty Q,
    letI* '(v1,v2), free := nd_seq (wp_operand e1) (wp_operand e2) in
      ∃ v',
        (eval_binop tu o (type_of e1) (type_of e2) ty v1 v2 v' ** T) ∧
        Q v' free)
    |-- wp_operand (Ebinop o e1 e2 ty) Q.
```

Evaluation order

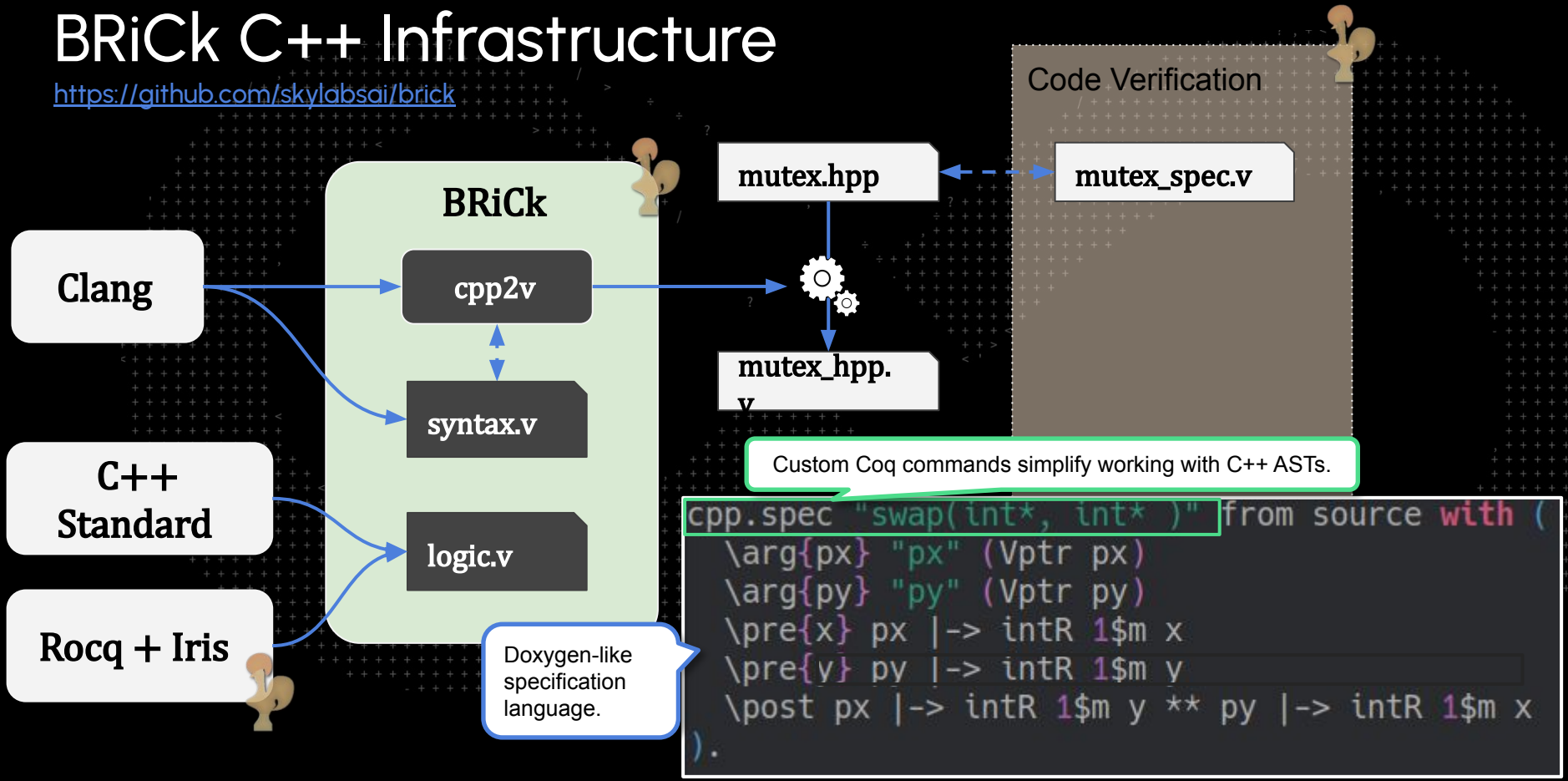
BRiCk C++ Infrastructure

https://github.com/skylabsai/brick



BRiCk C++ Infrastructure

https://github.com/skylabsai/brick



Mutex Specifications

Specification captures the ownership transfer.

- Mutex protects an invariant.
- **`enter()`** transfers resources to the caller.
- **`exit()`** transfers resources to the mutex.

```
cpp.spec "Sys::Mutex::enter()" as enter
  with (λ (this : ptr) ⇒
    \nova pd ec
    \persist{γ P} mutex.I (Some γ) P
    \prepost{q} this |-> mutex.R pd (Some γ) q
    \pre mutex.contender γ 1
    \post[Vbool true] P ** mutex.locked γ).
cpp.spec "Sys::Mutex::exit()" as exit
  with (λ (this : ptr) ⇒
    \nova pd ec
    \persist{γ P} mutex.I (Some γ) P
    \prepost{q} this |-> mutex.R pd (Some γ) q
    \pre |> P ** mutex.locked γ
    \post[Vbool true] mutex.contender γ 1).
```

User-defined representation predicates encapsulate implementation details.

Trade contender for lock ownership

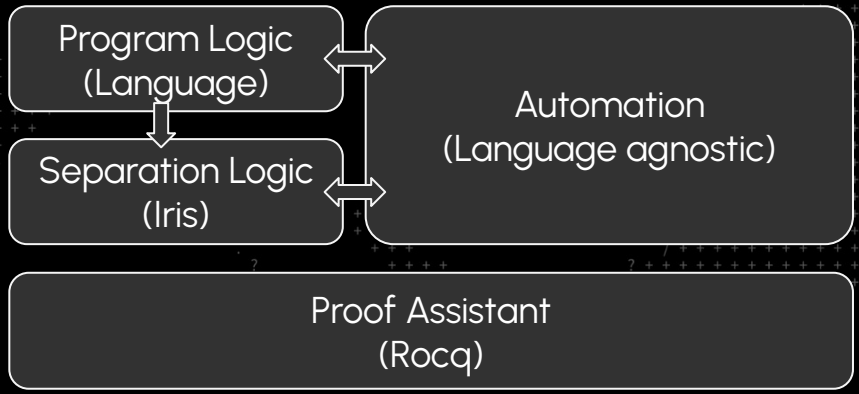
Return lock resources to get contender

Unrestricted use of Iris features for reasoning.

A Foundation for Software Verification

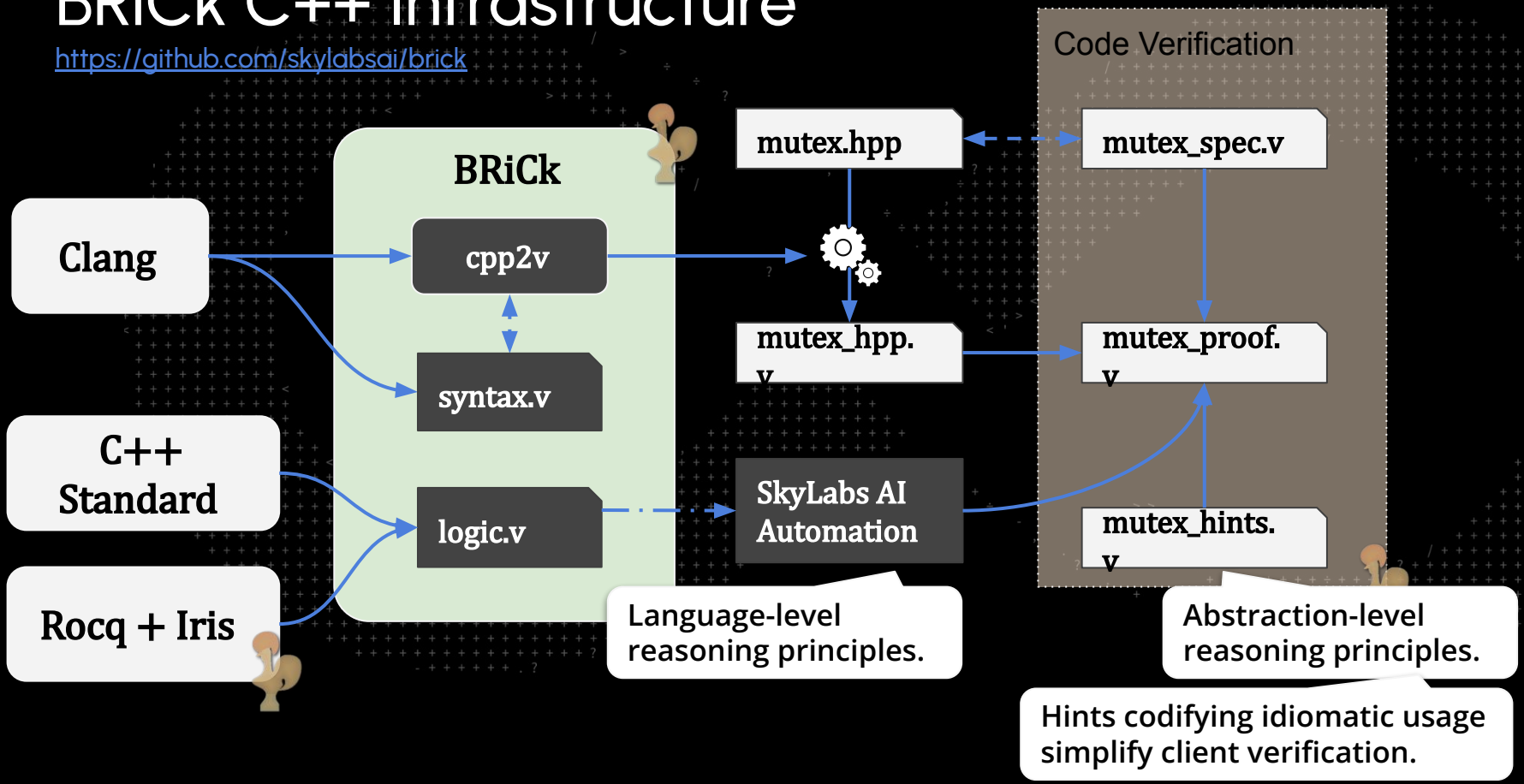
Automation is crucial for scale

- Iris ships with good proof support via Iris Proof Mode (IPM).
  - Excellent for doing sophisticated low-level reasoning
- **Powerful "follow your nose" automation** to scale to more code.
  - Handles the "administrative stuff" so we can focus on the meat of the proof.
  - Instantiating existentials, blasting through WPs, etc.



BRiCk C++ Infrastructure

https://github.com/skylabsai/brick



Verification with BRiCk

Language-level Reasoning

Hints capture reasoning principles for the automation to **apply automatically**.

- "Forward" hints work on premises
- "Backward" hints work in the conclusion
- "Cancellation" hints connect premises and conclusions.
- "Learnable" hints learn pure facts from conjunctions.

```
Axiom wp_lval_deref : ∀ ty e Q,
    letI* v , free := wp_operand e in
    ∃ p, [| v = Vptr p |] **
      reference_to (erase_qualifiers ty) p **
      Q p free
    ⊢ wp_lval (Ederef e ty) Q.
```

"One-liners" convert lemmas into a form usable by the core automation.

```
Definition  wp_operand_addrof_B :=[BWD] @wp_operand_addrof.
```

SkyLabs AI Automation

Extend the automation with BRiCk's language-level reasoning principles.

Verification with SL Automation

Reusable hints and hint infrastructure tames the verbosity of low-level proofs.

- Reduce overhead for "follow your nose" proofs.
- Make it tractable for non-experts to **maintain** & (sometimes) **develop** complex proofs.

600 tokens

```
Lemma create_pd_raw2safe_ipm : spec_entails raw.create_pd_spec safe.create_pd_spe
Proof.
  iIntros (vs K) ⇒ /=.
  iDestruct 1 as
    (y_caller_ec y_caller_obj y_caller_hst op sel_target sel_owner_pd y_owner_pd pd_rights q_pd ty) "R".
  iDestruct "R" as (→) "(ECobj & EChst & %HpdRights & CapOwn & CapTgt & %tyS & SP & Tail)".
  iExists _, _, _, (λ s, K (status.to_V s)), _, _, _. iFrame "ECobj EChst".
  iSplit; first done.
  iSplitL; last by iIntros (?) "$".
  iApply (resolve_sel_rights_intro_weak_sequential HpdRights with "CapOwn [-]"); iIntros "CapOwn /=".
  rewrite tyS.
  rewrite /create_pd.create_obj /create_pd.create_hst
          /create_pd.create_first_space /create_pd.create_pio /=.
  destruct ty.
  1, 4, 5 : (** PD, GST, DMA *)
    iApply (create_sel_intro_weak_sequential with "CapTgt");
      repeat iSplit; iIntros; iApply "Tail" ⇒ /=; eauto with iFrame.
  1, 2 : (** OBJ, HST *)
    iApply (do_reserve_spc_success_intro_weak_sequential with "SP");
      iIntros "SP";
      iApply (create_sel_cap_intro_weak_sequential with "CapTgt");
      (repeat iSplit; [..|iIntros (y_spc)]; iIntros "Cap";
        iApply (do_reserve_spc_commit_or_cancel_intro_weak_sequential with "SP");
        iIntros "SP"; iApply "Tail" ⇒ /=; eauto with iFrame).
  (** PIO *)
  - iDestruct (x86_only_elim with "SP") as (IS_x86 y_pio y_hst) "[SP SH]".
    iApply (do_reserve_spc_failure_intro_weak_sequential with "SP").
    iIntros "SP".
    iApply x86_only_intro; [done|].
    iApply (do_read_pd_hst_intro_weak_sequential with "SH"). iIntros "SH".
    iApply (create_sel_cap_intro_weak_sequential with "CapTgt").
    repeat iSplit; [..|iIntros (y_spc)]; iIntros "Cap";
      iApply "Tail" ⇒ /=;
      rewrite -?(x86_only_intro _ _ _ IS_x86);
      eauto 10 with iFrame.
  (** MSR *)
  - iDestruct (x86_only_elim with "SP") as (IS_x86) "_".
    iApply x86_only_intro; [done|].
    iApply (create_sel_intro_weak_sequential with "CapTgt");
      repeat iSplit; iIntros; iApply "Tail" ⇒ /=;
      rewrite -?(x86_only_intro _ _ _ IS_x86);
      eauto with iFrame.
Qed.
```

BlueRock

Several reusable hints later...

83 tokens

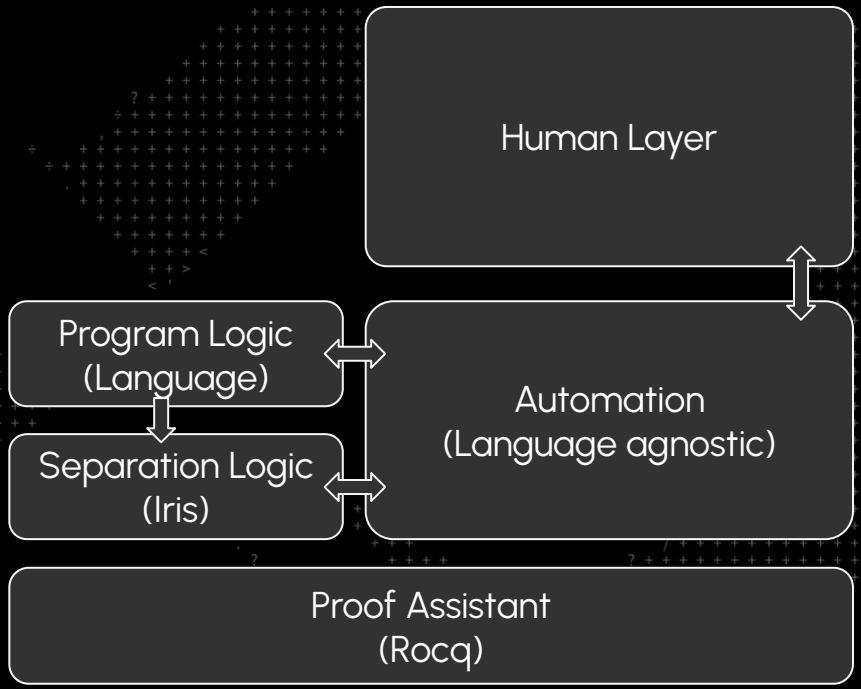
```
Lemma create_pd_raw2safe : spec_entails raw.create_pd_spec saf
Proof.
  iIntros (vs K) "/=".
  rewrite /create_pd.spec /create_pd.success
          /create_pd.create_obj /create_pd.create_hst
          /create_pd.create_first_space /create_pd.create_pio /=.
  work.
  ren_hyp ty spc.type.t.
  ren_hyp tyS (spc.type.of_N _ = Some ty).
  rewrite tyS.
  case_match; work.
  all: eauto.
Qed.
```

BlueRock

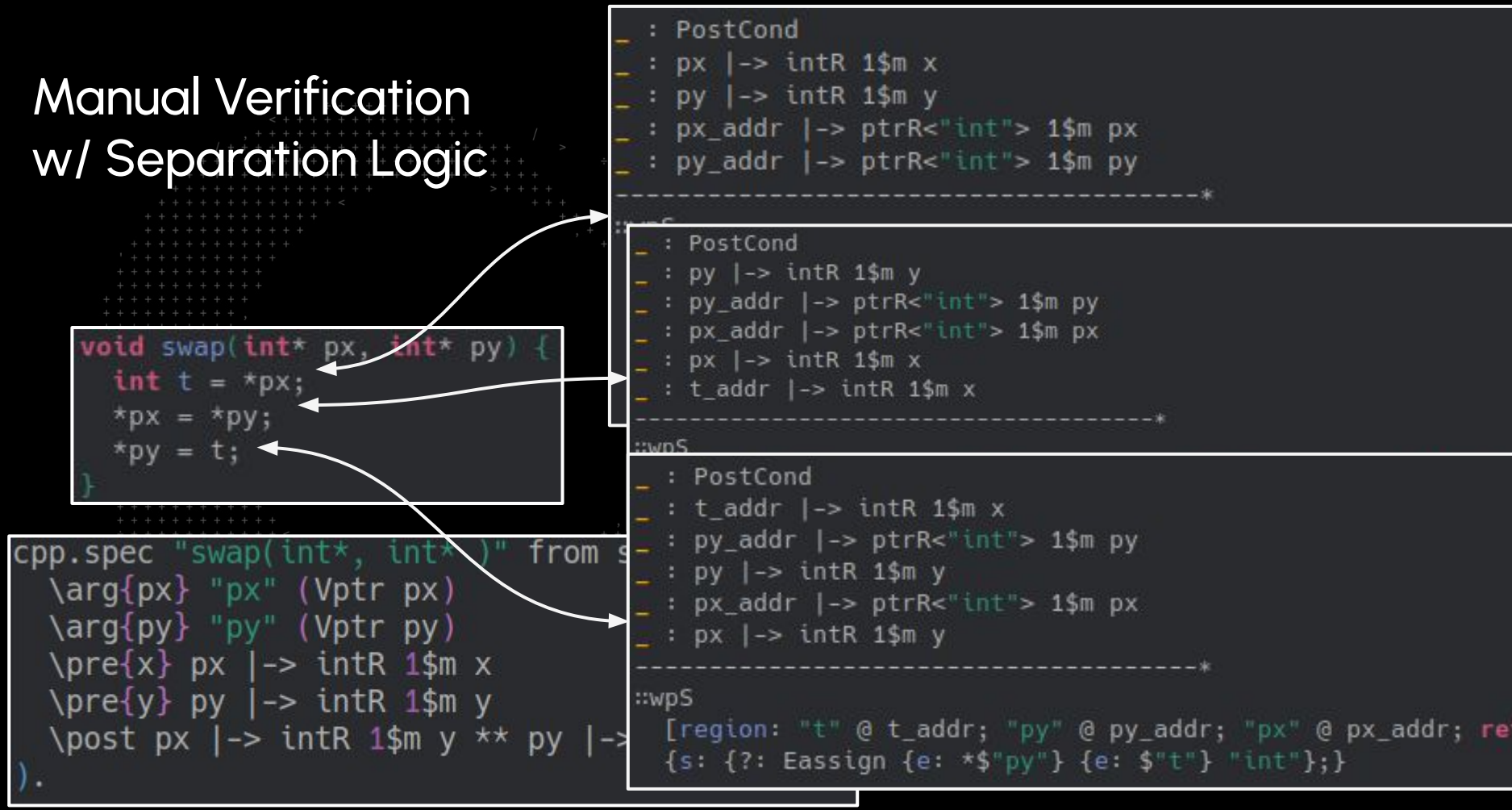
A Foundation for Software Verification

Humans drive the verification by:

- Providing insights when automation gets stuck, e.g. tactics
- Extending automation to handle repetitive reasoning.



Manual Verification w/ Separation Logic



Algorithmic

Algorithmics are enough to solve many problems.

- Most code is not very complex, but it is big.

Features such as:

- Straight-line code
- First-order function calls, even indirect
- Reconciling abstractions
- Many array access patterns

The "bread and butter" of business logic.

```
void swap(int* px, int* py) {
  int t = *px;
  *px = *py;
  *py = t;
}
```

```
cpp.spec "swap(int*, int* )" from source with (
  \arg{px} "px" (Vptr px)
  \arg{py} "py" (Vptr py)
  \pre{x} px |-> intR 1$m x
  \pre{y} py |-> intR 1$m y
  \post px |-> intR 1$m y ** py |-> intR 1$m x
).
```

```
Lemma swap_ok : verify[source] "swap(int*, int* )".
Proof. verify_spec. go. Qed.
```

Algorithmic

Purely algorithmic verification is not complete. 🤯

Classically difficult problems

- Loop invariant inference
- Concurrent invariants

SkyLabs' automation is designed to be conservative & predictable.

When we need smarts and intuition, we move up the stack.

```
int loop() {
  int i = 0;
  while (i < 10) {
    i++;
  }
  return i;
}
```

```
cpp.spec "loop()" from source with (
  \post[Vint 10] emp
).
```

```
Lemma loop_ok : verify[source] "loop()".
Proof using MOD.
  verify_spec; go.
  (* specify the loop invariant *)
```

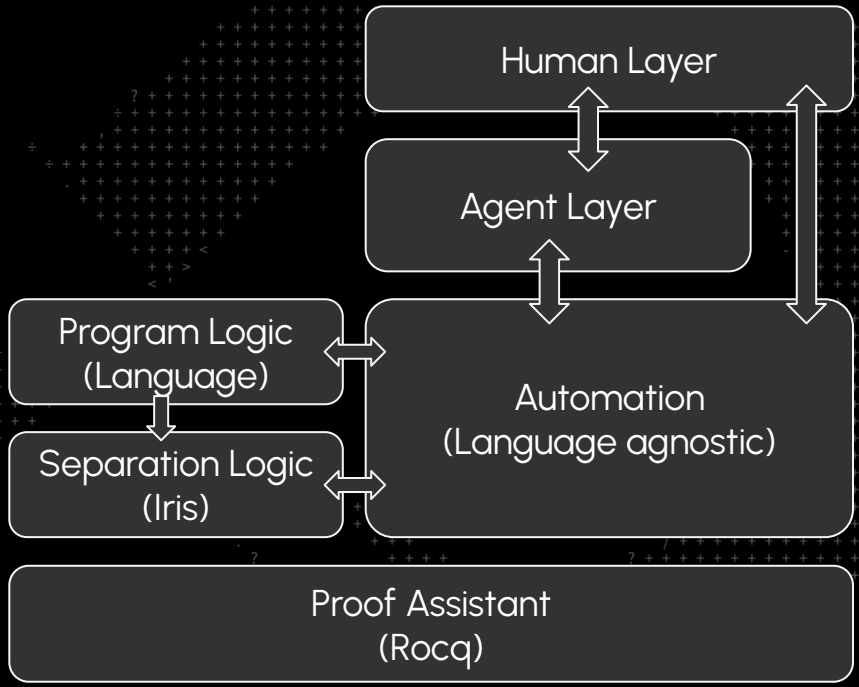
```
_ : PostCond
_ : i_addr |-> intR 1$m 0
-----------------------------------*
::wpS
  [region: "i" @ i_addr; return {?: "int"}]
  {s: while (($"i" < #10)) {
        $"i"++;
        // end block
      }}
```

The AI Layer

Human ↔ AI collaboration

- Tight loop integration, e.g. co-pilot.
- Shifting towards agentic delegation.

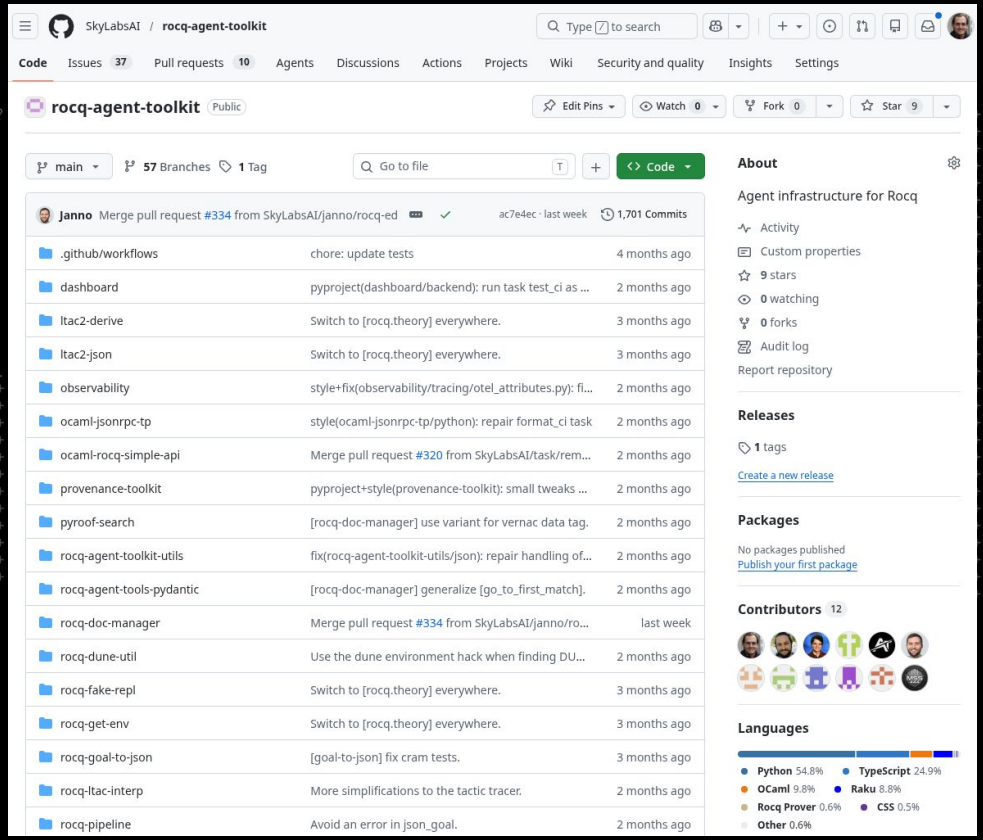
Need hardened tools for Agent↔Rocq interaction



Aside: Rocq Agent Toolkit

Provides tools and libraries for agents to interact with Rocq.

- Tools for inspecting the Rocq environment
- `rocq-ed` – command line interface for Rocq document interaction
- `rocq-doc-manager` – JsonRPC2 interface to Rocq w/ python bindings
- `pyroof-search` – algorithmic proof search built around "strategies" (e.g. Beam, MTS, etc)
- `rocq-agent-tools-pydantic` – Rocq tools for pydantic-ai
- `rocq-pipeline` – infrastructure for *narrow* agent evaluation, e.g. verify this lemma

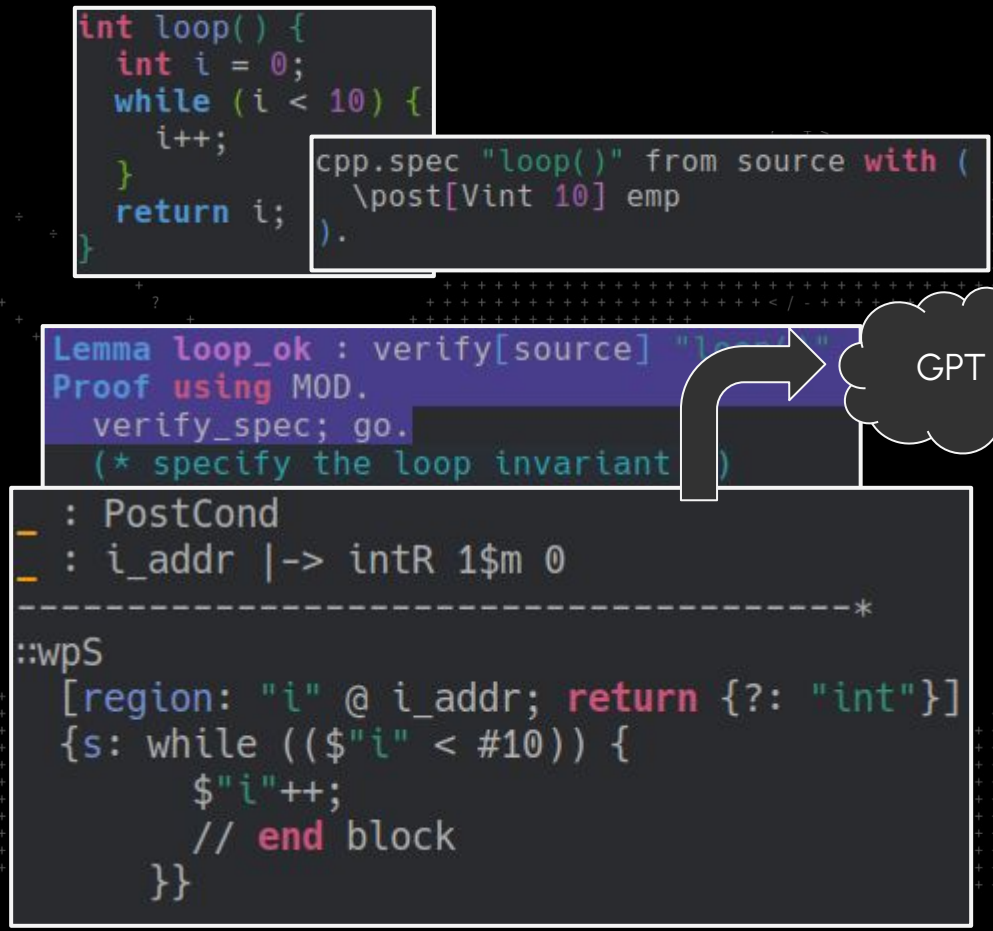


https://github.com/skylabsai/rocq-agent-toolkit

AI Oracles

Verification algorithms query agents as oracles for the "hard" problems.

- Avoid asking agents the routine questions.
- Specialize agents and prompts to specific problems:
  - different loop shapes (`for`, `while`, `for-each`),
  - reconciling abstractions, etc.
- General "next tactic prediction"



AI Oracles

Verification algorithms query agents as oracles for the "hard" problems.

- Avoid asking agents the routine questions.
- Specialize agents and prompts to specific problems:
  - different loop shapes (`for`, `while`, `for-each`),
  - reconciling abstractions, etc.
- General "next tactic prediction"

Suitable for more problems, but necessary to engineer context for each agent call.

```
int loop() {
  int i = 0;
  while (i < 10) {
    i++;
  }
  return i;
}
```

```
cpp.spec "loop()" from source with (
  \post[Vint 10] emp
).
```

```
Lemma loop_ok : verify[source] "loop"
Proof using MOD.
  verify_spec; go.
  (* specify the loop invariant *)
  wp_while (λ ρ ⇒
    ∃ i, _local ρ "i" |-> intR 1$m i **
         [| (i ≤ 10)%Z |])%I.
  go.
Qed.
```

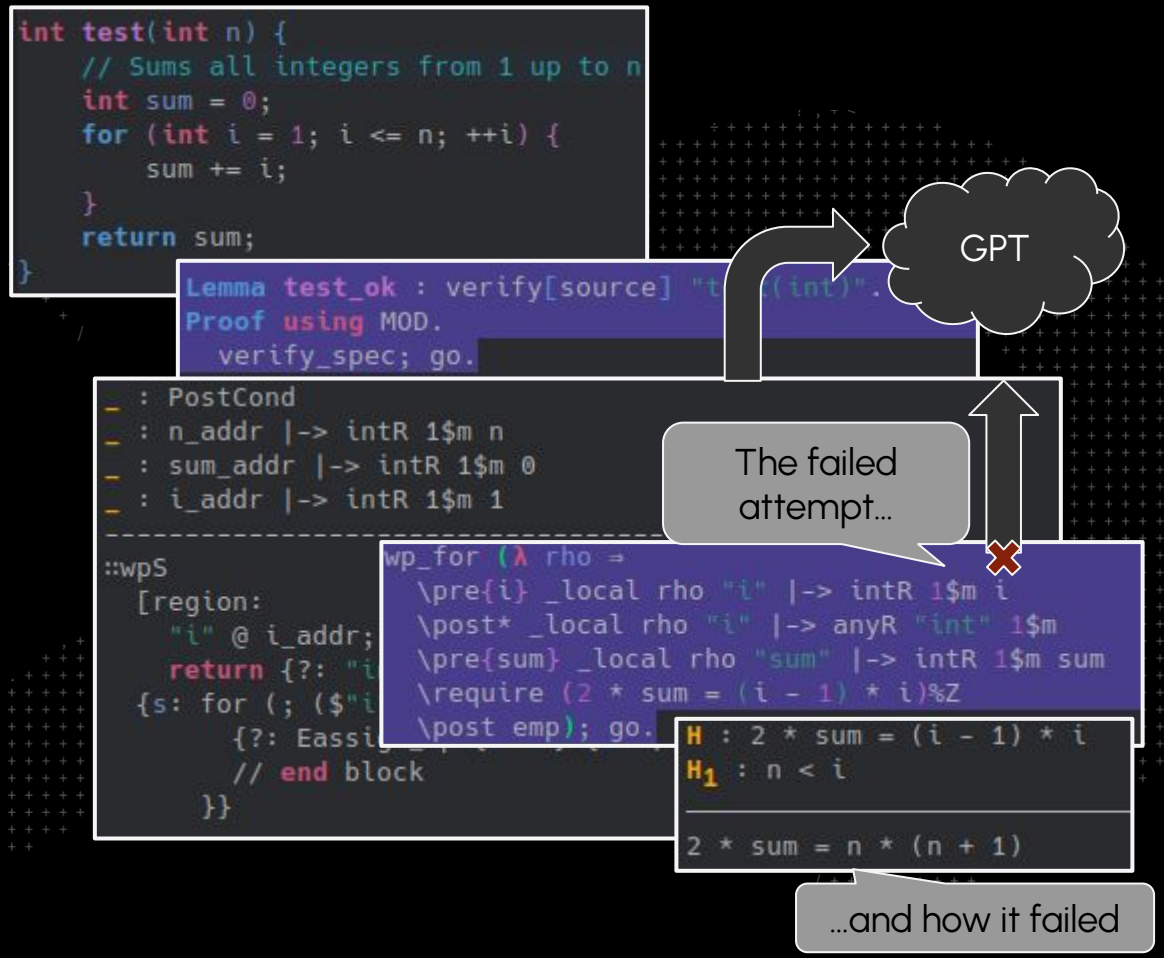
Limitations
AI Oracles

Narrow Agentic

Crucial for models to learn from their mistakes.

- Subsequent model calls need **access to failed attempts** to refine their actions.

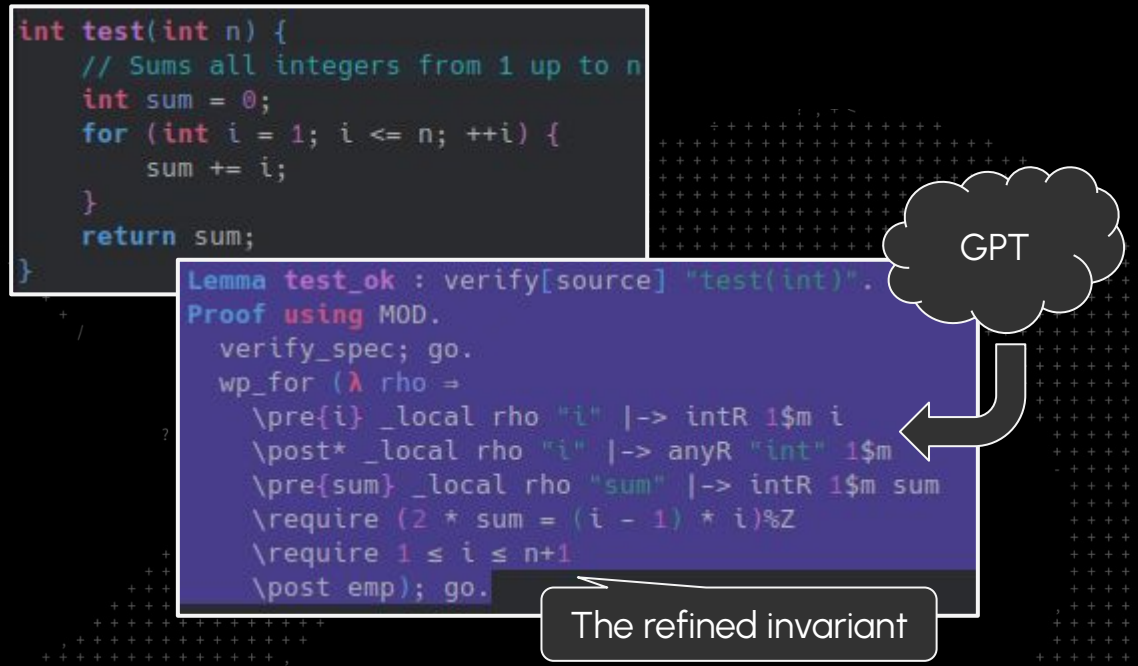
Context engineering is crucial!



Limitations
AI Oracles

Crucial for models to learn from their mistakes.

- Subsequent model calls need **access to failed attempts** to refine their actions.



Context engineering is crucial!

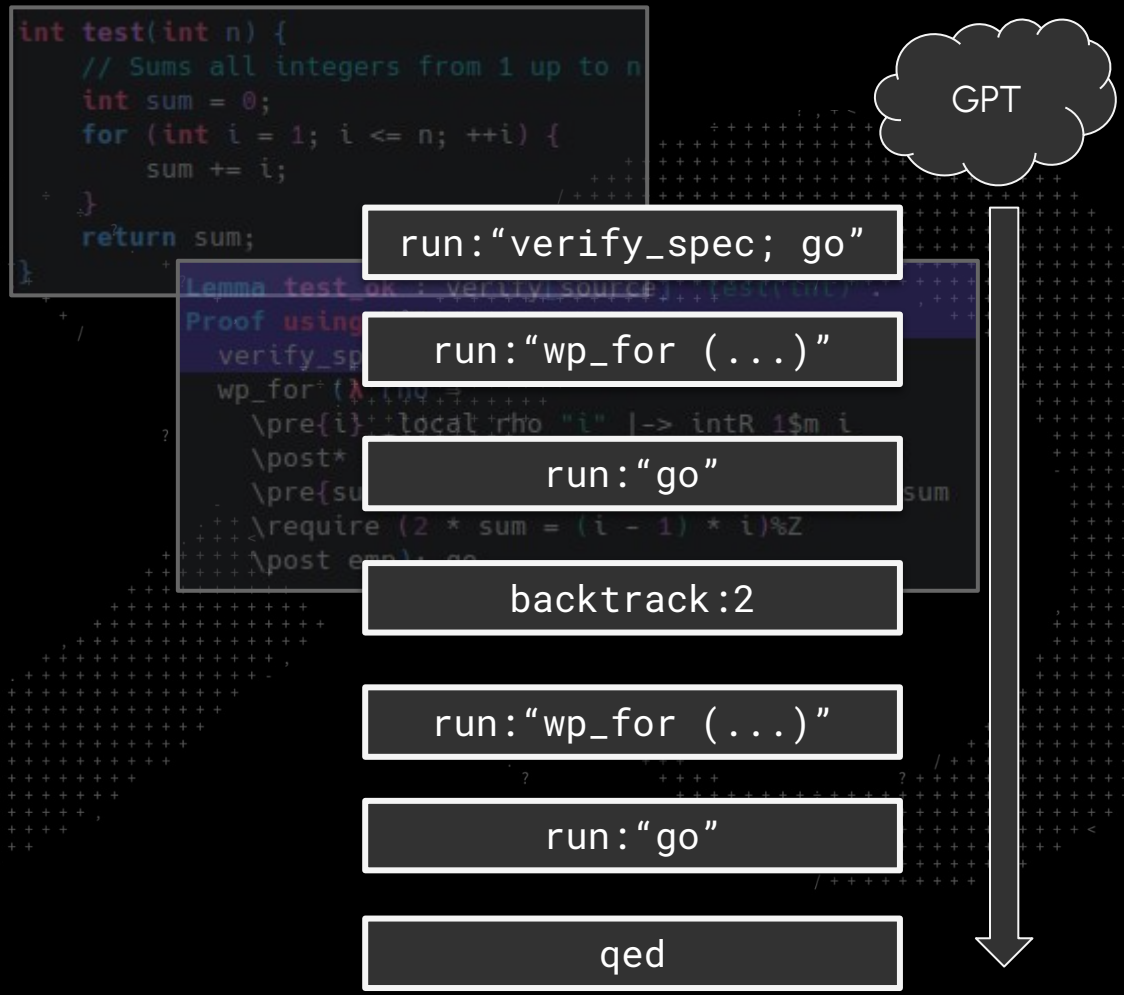
Narrow Agentic

Put agents in control **for a single proof**

- Constrained and engineered context
- **Longer-lived agent** interacts with Rocq through tools, including backtracking.
  - Tool selection is the agent's responsibility.
- **Enabled by more powerful models**
- Easy to prevent cheating during eval

More tokens, but more flexibility.

Built on rocq-agent-tools-pydantic.



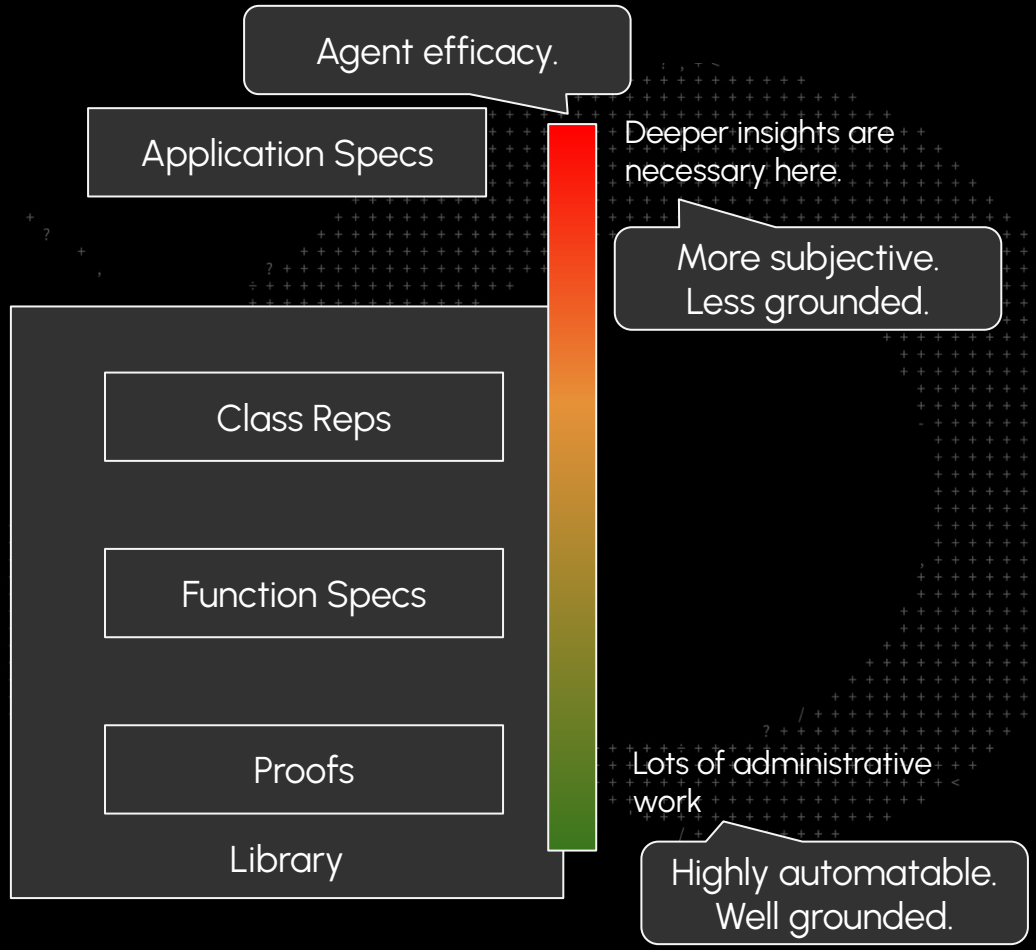
The Future

Spec-driven verification.

- Humans focus on specs & high-level abstractions.
- Agents fill in the proofs.
- **Good specifications signal good abstractions!**

Farther ahead (but not too far?)

- Expanding agent capabilities:
  - Writing specs and representation predicates



Questions

Authors
Lennart Beringer
Jasper Haag
Rodolphe Lepigre
Gregory Malecha
Rayan Soban
Ehtesham Zahoor

Additional Contributors
Janno Kaiser
Simon Hudon
Paolo G. Giarrusso
Hai Dang (BlueRock Security)
Abhishek Anand (Category Labs)

- Separation Logic as a foundation for verification
  - Language-agnostic
  - Full system (multi-program)
  - Concurrency
  - Refinement
- Tools for humans **& agents**
  - Rocq: general purpose proof assistant
  - Embedded logic **w/ automation**
- Algorithmic<->Agentic architectures