WHEN AR MET AI

CSLib
Building a Platform for AI-assisted Formal Verification in Lean

Clark Barrett

SAFE
Stanford | Center for AI Safety

CENTAUR
Stanford | Center for Automated Reasoning

Art credit: Elena Valentine Barrett

High Confidence Software and Systems Conference
May 12, 2026

Artificial Intelligence and Automated Reasoning

- Artificial Intelligence (AI) increasingly used in many domains
  - Powerful new capabilities
  - Good with messy, ill-formed problems
  - Data-driven
  - **No correctness guarantees (e.g., adversarial inputs and hallucinations)**

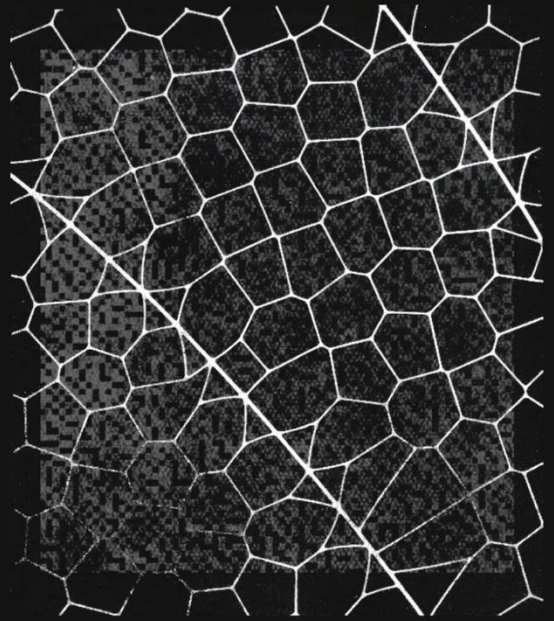
- Automated Reasoning (AR)
  - Computerized formal mathematical reasoning
  - Extensive use in industry to prove that systems are correct
  - Correctness guarantees

Emerging threat: Security in the age of AI

Project Glasswing

Securing critical software for the AI era

Continue reading



What about
Automated Reasoning / Formal Methods?

Risks of GenAI Workshop

- Google, June 27, 2023
- Lots of discussion about risks and worries

- If GenAI can find security vulnerabilities, our systems will be less safe and secure

- Me: Well, there is a solution…it’s called formal verification

- Silence…

Traditional Response to Formal Verification

1. It requires monumental effort from human experts
2. Automated techniques are too weak or don’t scale
3. Not economically viable

Guaranteed Safe AI Summit

- San Francisco, March 8, 2025
- Lots of discussion about risks and worries

- Almost every talk mentions Formal Methods

- Me: Silence…

AI as a catalyst for formal methods

- People are concerned about a ~~future~~ *present* in which:
  - AI systems create mountains of buggy software
  - AI systems dramatically increase the capabilities of black hat hackers
- AI Safety advocates are looking to formal methods to ensure safe AI
- A lot of excitement around Lean and AI-assisted theorem proving

How do we get there?

AI as a catalyst for formal methods

- AI capabilities advancing at a furious pace
  - Reasoning about code
  - Writing Lean proofs

- But there is a big difference between writing Lean proofs and reasoning about code

- How do we bridge this gap?

Introducing CSLib

CSLib

- Inspired by success of Mathlib

- Goals include:
  - Formalizing CS theory
  - A platform for reasoning about code
  - Repository of verified code
  - AI integration

CSLib steering committee

Clark Barrett

Stanford and Amazon

Swarat Chaudhuri

UT Austin and Google DeepMind

Leo de Moura

Sr Principal Scientist

Amazon

Jim Grundy

Sr Principal Scientist

Amazon

Pushmeet Kohli

VP of Research

Google DeepMind

Fabrizio Montesi

Universiity of Southern Denmark

CSLib technical leads

Alexandre Rademaker

Fundação Getulio Vargas and Renaissance Philanthropy

Sorrachai Yingchareonthawornchai

ETH-ITS Zurich

Lean is an open-source programming language and proof assistant that enables correct, maintainable, and formally verified code

Install Learn

Powerful automation | Mathematics

```
-- 'Grind' efficiently manages complex pattern matching and
-- case analysis beyond standard tactics.

example (x : Nat) : 0 < match x with
  | 0   => 1
  | n+1 => x + n := by
  grind

-- Automatically solves systems of linear inequalities.

example (x y : Int) :
    27 ≤ 11*x + 13*y → 11*x + 13*y ≤ 45
    → -10 ≤ 7*x - 9*y → 7*x - 9*y > 4 := by
  grind
```

Grind is a powerful tool that can help you prove theorems quickly and efficiently.

Trustworthy

Lean's minimal trusted kernel guarantees absolute correctness in mathematical proof, software and hardware verification.

Powerful

From elementary concepts to cutting-edge research, Lean's expressive language and extensive built-in tools let users focus on the big picture rather than routine details.

Extensible

Lean's metaprogramming capabilities enable users to extend the language with domain-specific notations and new proof automation techniques.

Who is CSLib for?

CSLib: An open-source Lean 4 library, built through a community effort, formalizing computer science.

Target users

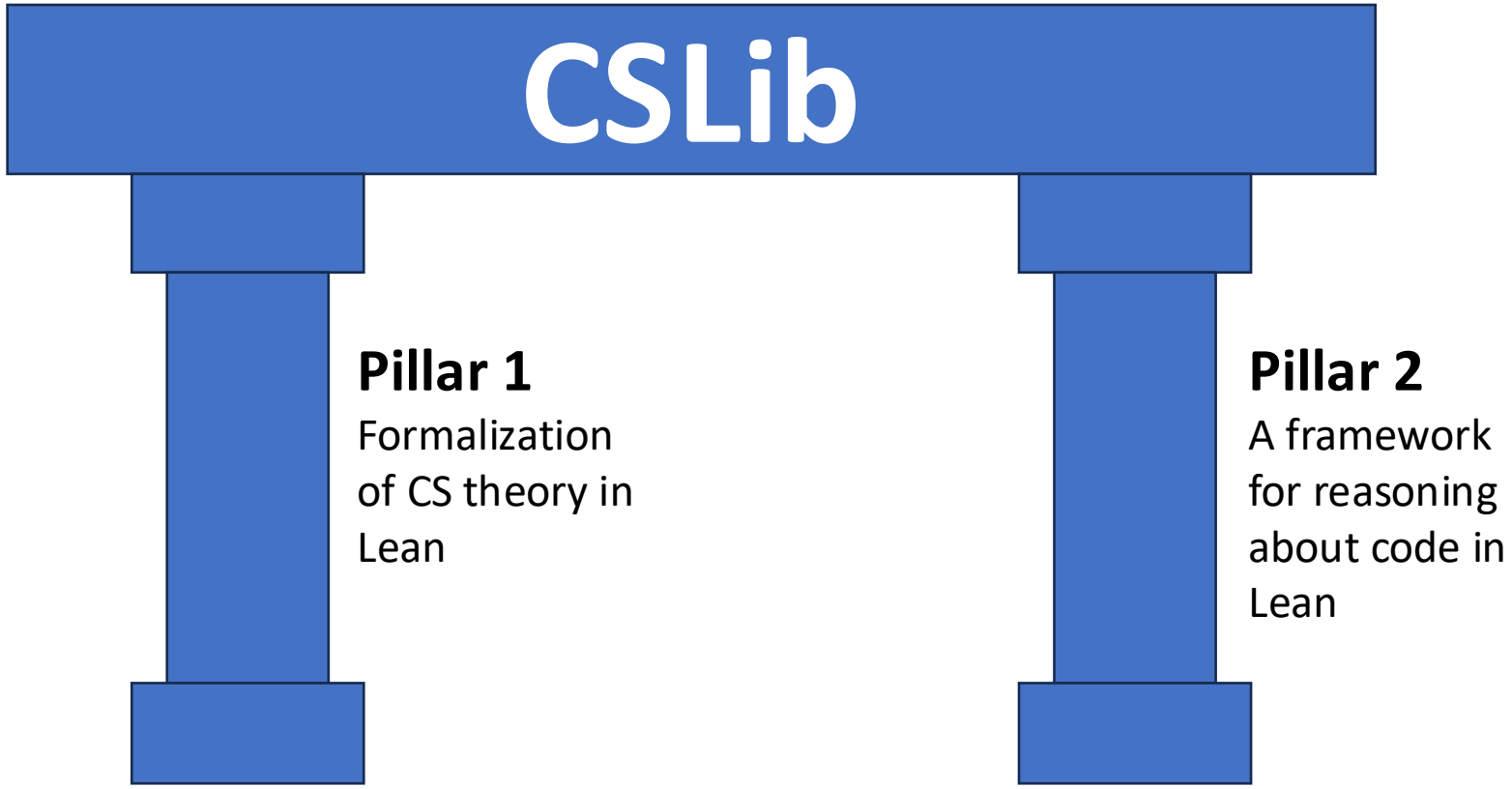
Educators who want to inject formal methods into computer science curricula.

Formal methods researchers and practitioners who want to verify algorithms and systems in Lean

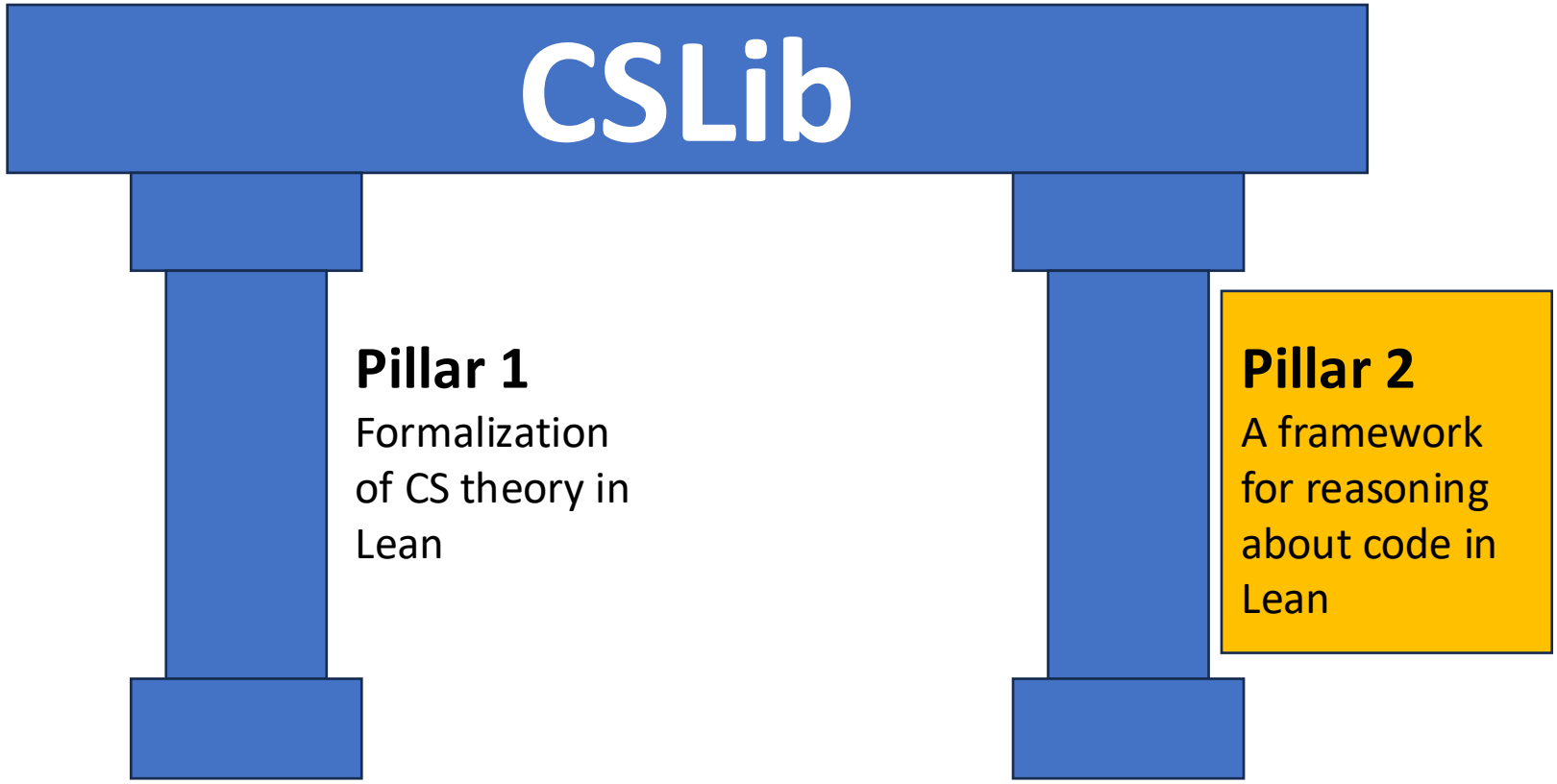
AI researchers who want to build AI tools for programming and reasoning

**All clip art generated with Gemini*

What is CSLib?



What is CSLib?



Deductive Program Verification

```
1  fn findmax(nums: Vec<u32>) → (ret: u32)
2  requires
3      nums@.len() > 0
4  ensures
5      forall|i: int| 0 ⩽ i < nums@.len() ⟹ nums@[i] ⩽ ret,
6      exists|i: int| 0 ⩽ i < nums@.len() && nums@[i] == ret
7  {
8      let mut max = nums[0];
9      let mut i = 1;
10     while i < nums.len()
11     invariant
12         1 ⩽ i ⩽ nums.len(),
13         forall|j: int| 0 ⩽ j < i ⟹ nums@[j] ⩽ max,
14         exists|j: int| 0 ⩽ j < i && nums@[j] == max
15     {
16         if nums[i] > max {
17             max = nums[i];
18         }
19         i += 1;
20     }
21     max
22 }
23
24
```

Specifications: Logical Formulas encoding functionality

Proof Code: Proof Constructs that help prove the specification

Deductive Program Verification

```
1  fn findmax(nums: Vec<u32>) → (ret: u32)
2  requires
3      nums@.len() > 0
4  ensures
5      forall|i: int| 0 ≤ i < nums@.len() ⟹ nums@[i] ≤ ret,
6      exists|i: int| 0 ≤ i < nums@.len() && nums@[i] == ret
7  {
8      let mut max = nums[0];
9      let mut i = 1;
10     while i < nums.len()
11     invariant
12         1 ≤ i ≤ nums.len(),
13         forall|j: int| 0 ≤ j < i ⟹ nums@[j] ≤ max,
14         exists|j: int| 0 ≤ j < i && nums@[j] == max
15     {
16         if nums[i] > max {
17             max = nums[i];
18         }
19         i += 1;
20     }
21     max
22 }
23
24
```

Specifications: Logical Formulas encoding functionality

Proof Code: Proof Constructs that help prove the specification

Verifier

Automatically Check: Correct or Incorrect

Computer

CSLib ecosystem

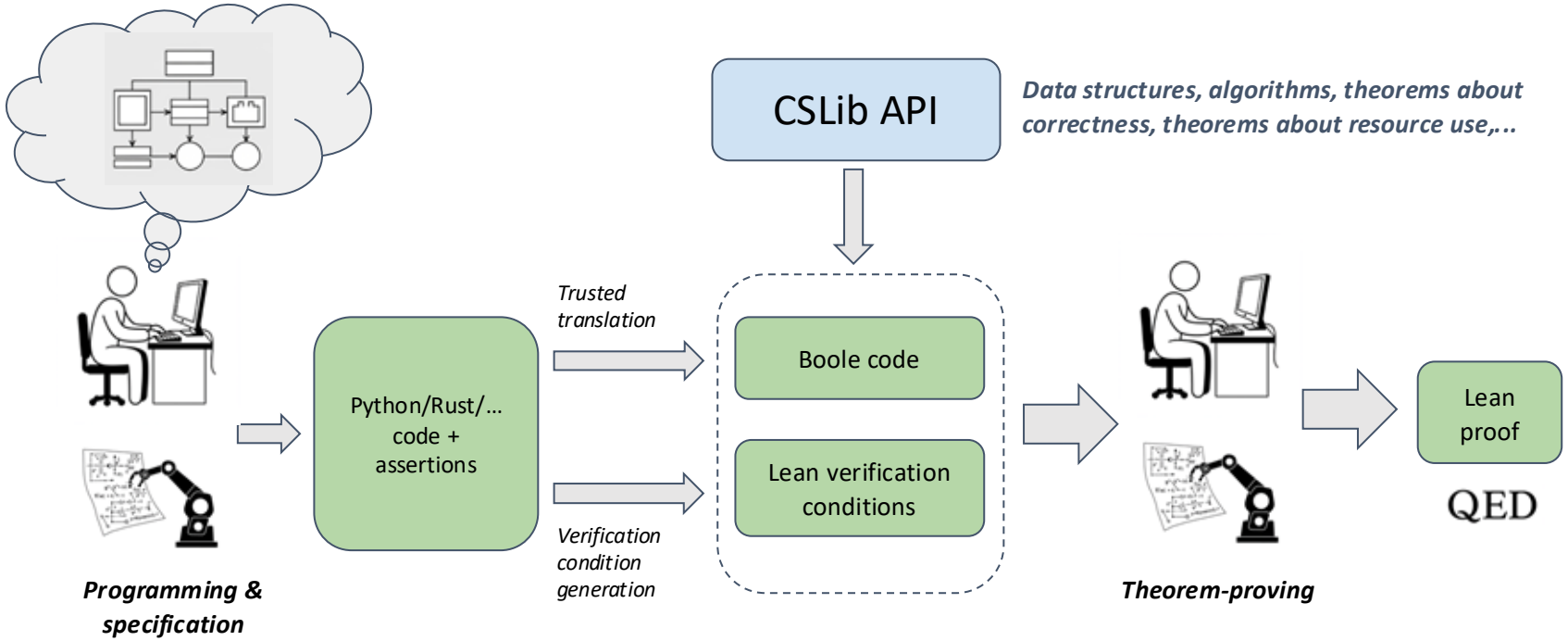
1. **Boole:** A verification-friendly intermediate language, embedded as a DSL in Lean
   - Defined using Amazon's Strata framework
   - Can be extended to handle new languages and features.

2. **CSLib:** Boole and Lean formalizations of all **general CS knowledge**
   - Algorithms and data structures in Boole, specs and proofs in Lean
   - CS-specific mathematics (e.g., complexity theory) in Lean

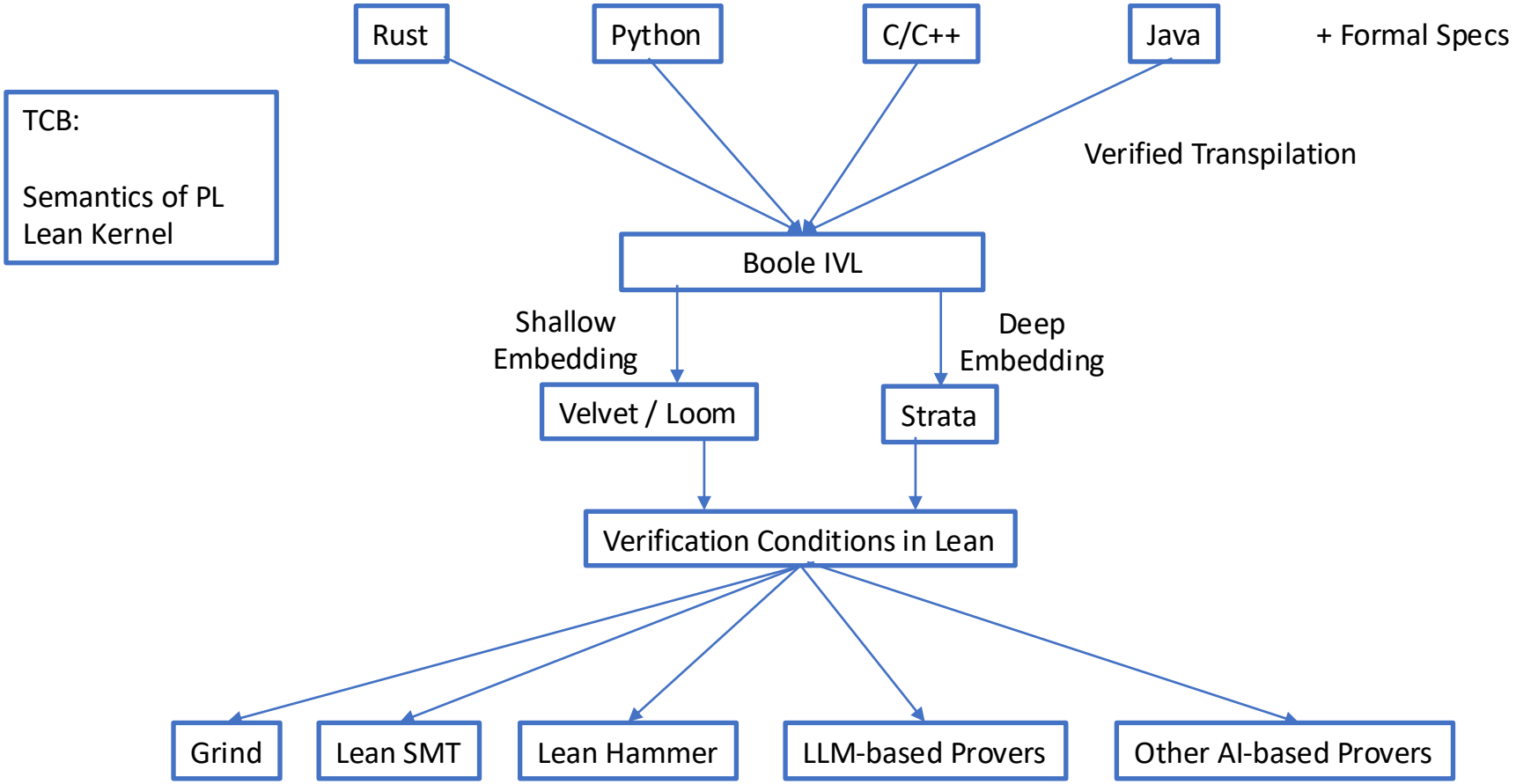
3. **Language embeddings:** Trusted translations from Rust, Python, C,... to Boole.

All code released under a permissive open-source license

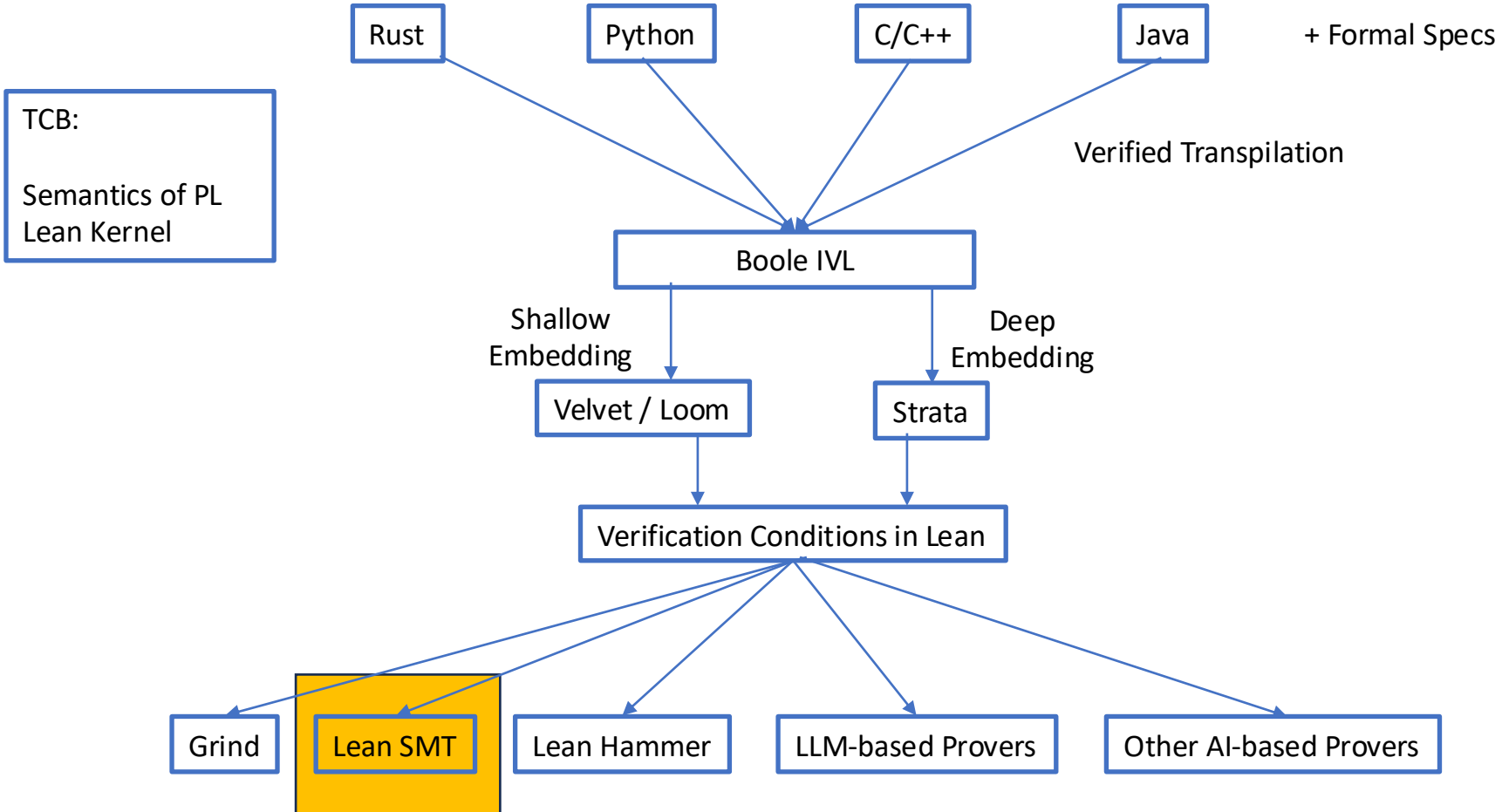
Usage scenario: Reasoning about code



CSLIB Pillar 2 Vision



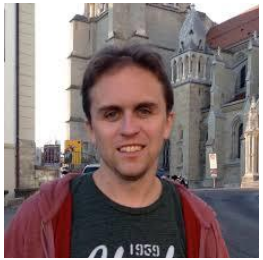
CSLIB Pillar 2 Vision



Lean SMT







Abdalrhman Mohamed
Stanford

Tomaz Mascarenhas
UFMG

Harun Khan
Stanford

Haniel Barbosa
UFMG

Andrew Reynolds
U Iowa

Yicheng Qian
Stanford

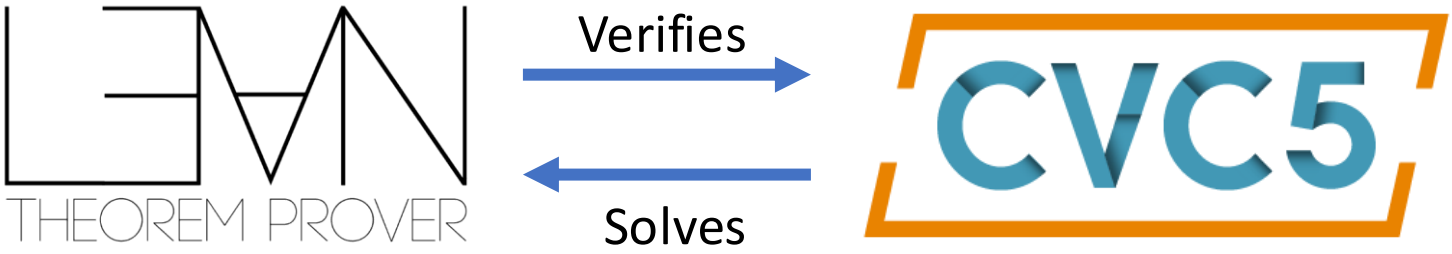
Cesare Tinelli
U Iowa

Lean-SMT: An SMT Tactic for Discharging Proof Goals in Lean, by Abdalrhman Mohamed, Tomaz Mascarenhas, Harun Khan, Haniel Barbosa, Andrew Reynolds, Yicheng Qian, Cesare Tinelli, and Clark Barrett, *37th International Conference on Computer Aided Verification (CAV '25)*

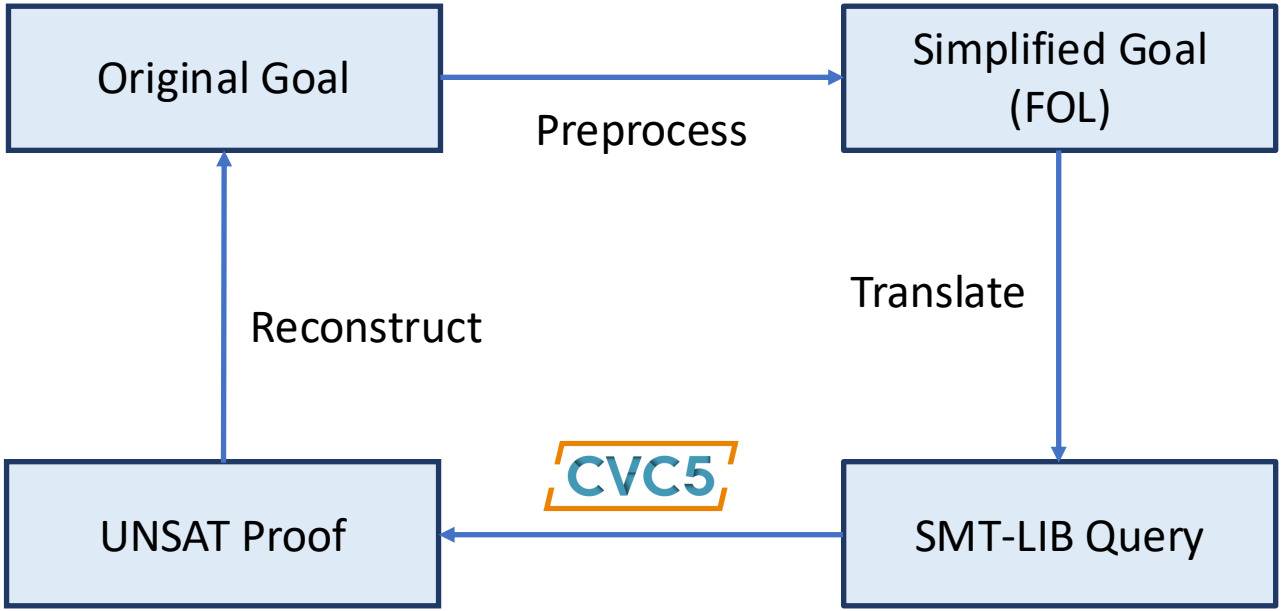
The Lean-SMT Tactic

Inspired by the success of SledgeHammer in Isabelle

A tactic that integrates cvc5 and Lean would benefit both worlds



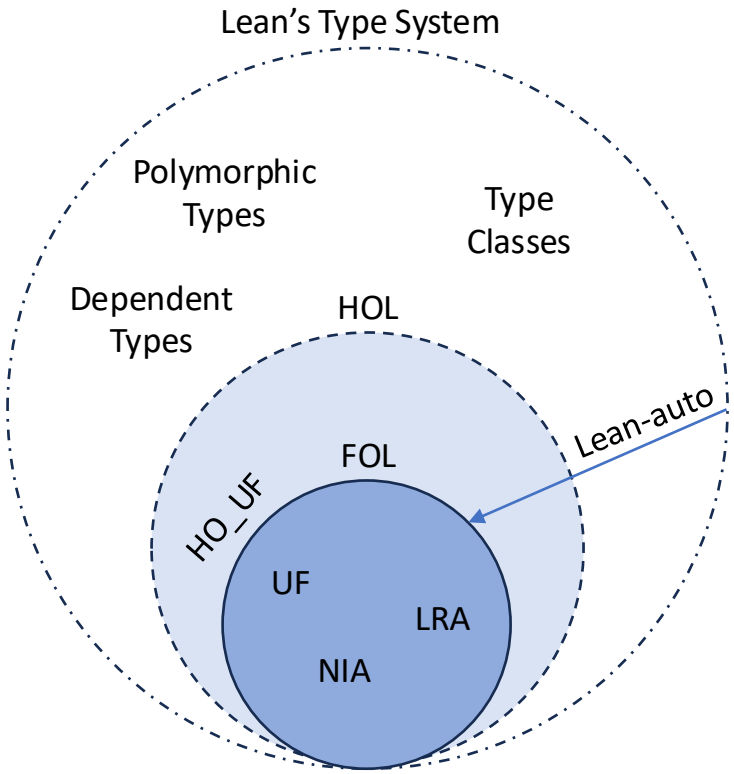
Tactic Workflow



Lean-SMT's Pipeline

Preprocessing | Translation | Solving | Reconstruction

- SMT-LIB theories are minimal.
- Do not support every operation and sort.
- Reduce papercuts:
  - Replace $p \leftrightarrow q$ with $p = q$
  - Embed `Bool` into `Prop`
  - Embed `Nat` into `Int`
  - Etc.
- What about Lean's complex type system?
- Utilize Lean-auto!



Lean-SMT's Pipeline

Preprocessing | Translation | Solving | Reconstruction

Original Lean Goal

```
⊢ ∀ (G : Type u) [Group G] (e : G),
    (∀ (a : G), e * a = a) ↔ e = 1
```

Identity element is unique in any group G!

Preprocessed Lean Goal

```
G: Type u
inst: Group G
e e': G
op: G → G → G
inv: G → G
one_mul: ∀ (a : G), op e a = a
inv_mul_cancel: ∀ (a : G), op (inv a) a = e
mul_assoc: ∀ (a b c : G),
 op (op a b) c = op a (op b c)

⊢ (∀ (a : G), (op e' a = a)) = (e' = e)
```

Lean-SMT's Pipeline

Preprocessing | **Translation** | Solving | Reconstruction

Preprocessed Lean Goal

```
G: Type u
inst: Group G
e e': G
op: G → G → G
inv: G → G
one_mul: ∀ (a : G), op e a = a
inv_mul_cancel: ∀ (a : G), op (inv a) a = e
mul_assoc: ∀ (a b c : G),
 op (op a b) c = op a (op b c)

⊢ (∀ (a : G), (op e' a = a)) = (e' = e)
```

Translation is sound because G is nonempty!

SMT-LIB Query

```
(declare-sort G 0)
(declare-const e G)
(declare-const |e'| G)
(declare-fun op (G G) G)
(declare-fun inv (G) G)
(assert (forall ((a G)) (= (op e a) a)))
(assert (forall ((a G))
  (= (op (inv a) a) e)))
(assert (forall ((a G) (b G) (c G))
  (= (op (op a b) c)
     (op a (op b c))))))
(assert (distinct
  (forall ((a G)) (= (op |e'| a) a)
  (= |e'| e))))
(check-sat)
```

Lean-SMT's Pipeline

Preprocessing → Translation → **Solving** → Reconstruction

SMT-LIB Query

```
(declare-sort G 0)
(declare-const e G)
(declare-const |e'| G)
(declare-fun op (G G) G)
(declare-fun inv (G) G)
(assert (forall ((a G)) (= (op e a) a)))
(assert (forall ((a G))
  (= (op (inv a) a) e)))
(assert (forall ((a G) (b G) (c G))
  (= (op (op a b) c)
     (op a (op b c)))))))
(assert (distinct
  (forall ((a G)) (= (op |e'| a) a)
  (= |e'| e))))
(check-sat)
```

cvc5 Proof (CPC)

```
(define @t1 () (eo::var "a" G))
(define @t2 () (op e @t1))
...
(assume @p1 (forall @t3 (= @t2 @t1)))
(assume @p2 (forall @t3 (= @t4 e)))
(assume @p3 @t8)
(assume @p4 (not (= (forall @t3 (= @t9 @t1))
                    (= |e'| e))))
(step @p5 :rule quant-merge-prenex
          :args ((= @t8 @t10)))
(step @p6 :rule eq_resolve
          :premises (@p3 @p5))
(step @p7 :rule eq-symm :args (@t4 e))
(step @p8 :rule cong :premises (@p7)
          :args ((forall ((a G)))))
...
```

Lean-SMT's Pipeline

Preprocessing → Translation → Solving → **Reconstruction**

cvc5 Proof (CPC)

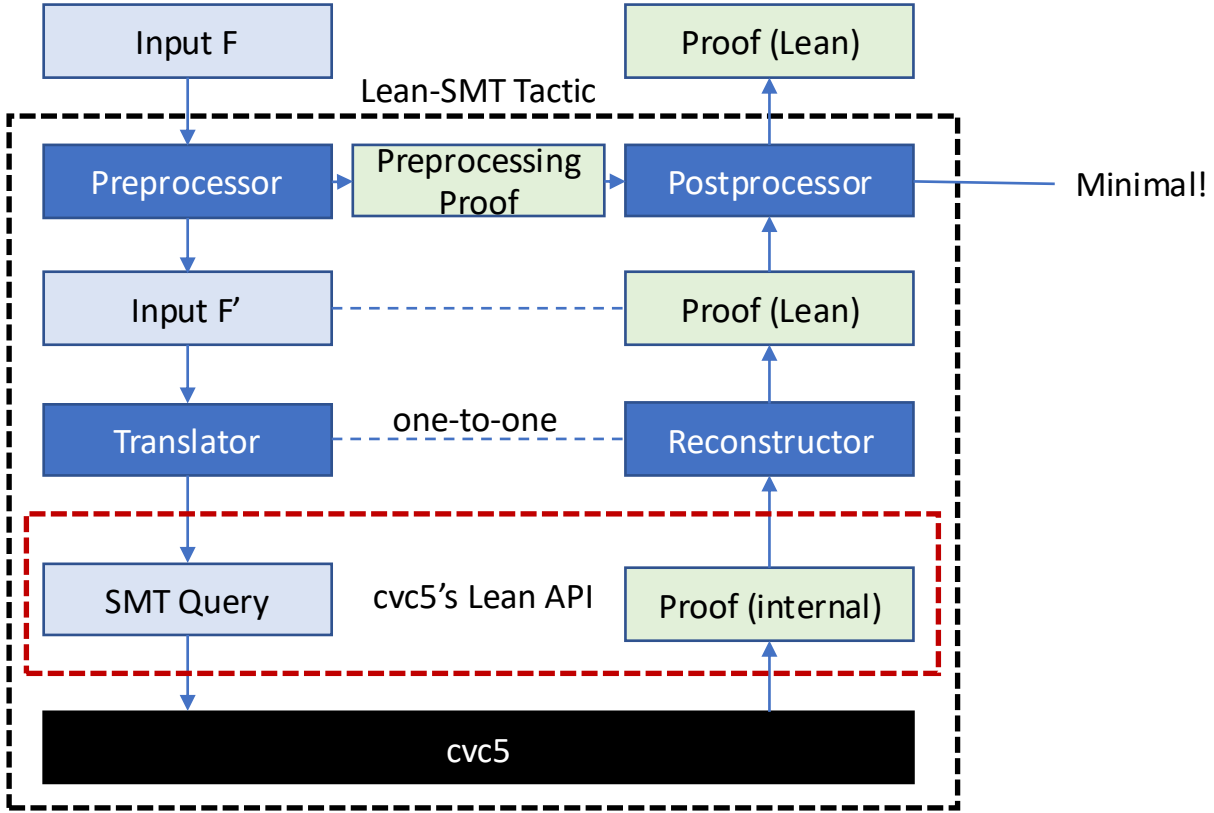
```
(define @t1 () (eo::var "a" G))
(define @t2 () (op e @t1))
...
(assume @p1 (forall @t3 (= @t2 @t1)))
(assume @p2 (forall @t3 (= @t4 e)))
(assume @p3 @t8)
(assume @p4 (not (= (forall @t3 (= @t9 @t1))
                    (= |e'| e))))
(step @p5 :rule quant-merge-prenex
          :args ((= @t8 @t10)))
(step @p6 :rule eq_resolve
          :premises (@p3 @p5))
(step @p7 :rule eq-symm :args (@t4 e))
(step @p8 :rule cong :premises (@p7)
          :args ((forall ((a G)))))
...
```

Lean Proof

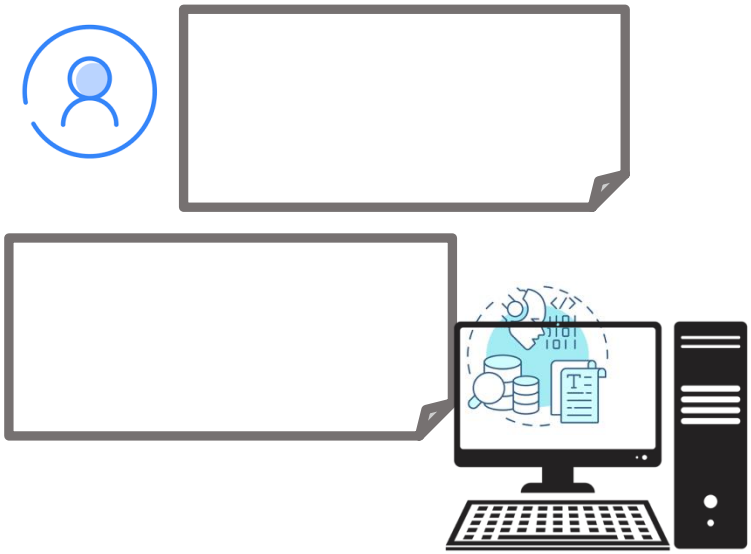
```
have a0 : e = op (inv e') e' := by assumption
have a1 : e' = op e' e' := by assumption
...
have s2 : inv e' = inv e' := Eq.refl (inv e')
have s3 : op e' e' = e' := Eq.symm a1
have s4 : op (inv e') (op e' e') =
          op (inv e') e' :=
  congr (congr (Eq.refl op) (Eq.refl (inv
e')))
        (Eq.symm a.17)
have s5 : op (inv e') e' = e := Eq.symm a0
have s6 : op (inv e') (op e' e') = e :=
  Eq.trans (congr (congr (Eq.refl op)
                         (Eq.refl (inv e')))
             (Eq.symm a1)) (Eq.symm a0)
have s7 : e' = e' := Eq.refl e'
...
```

Check with Lean's Kernel!!!

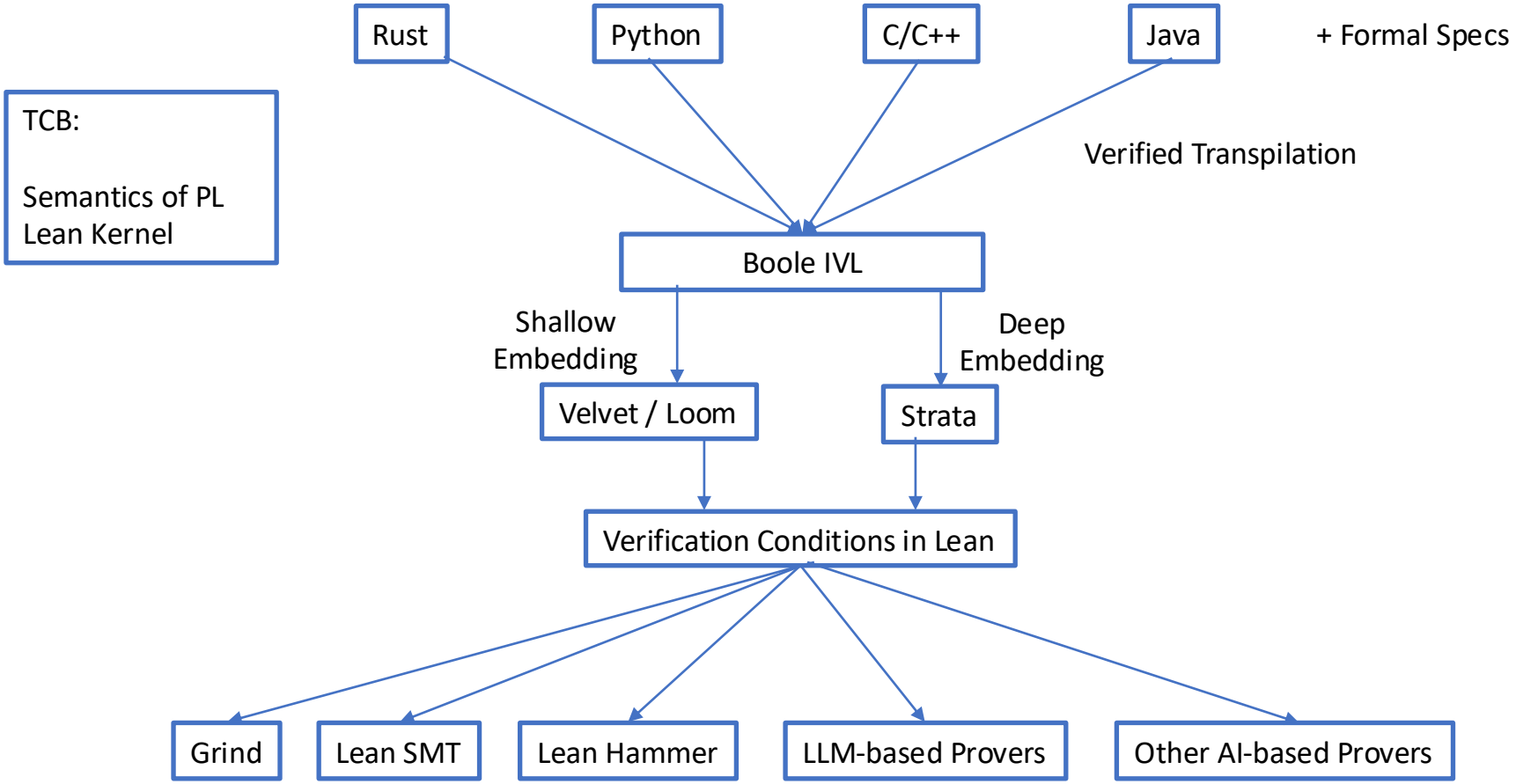
Overall Architecture



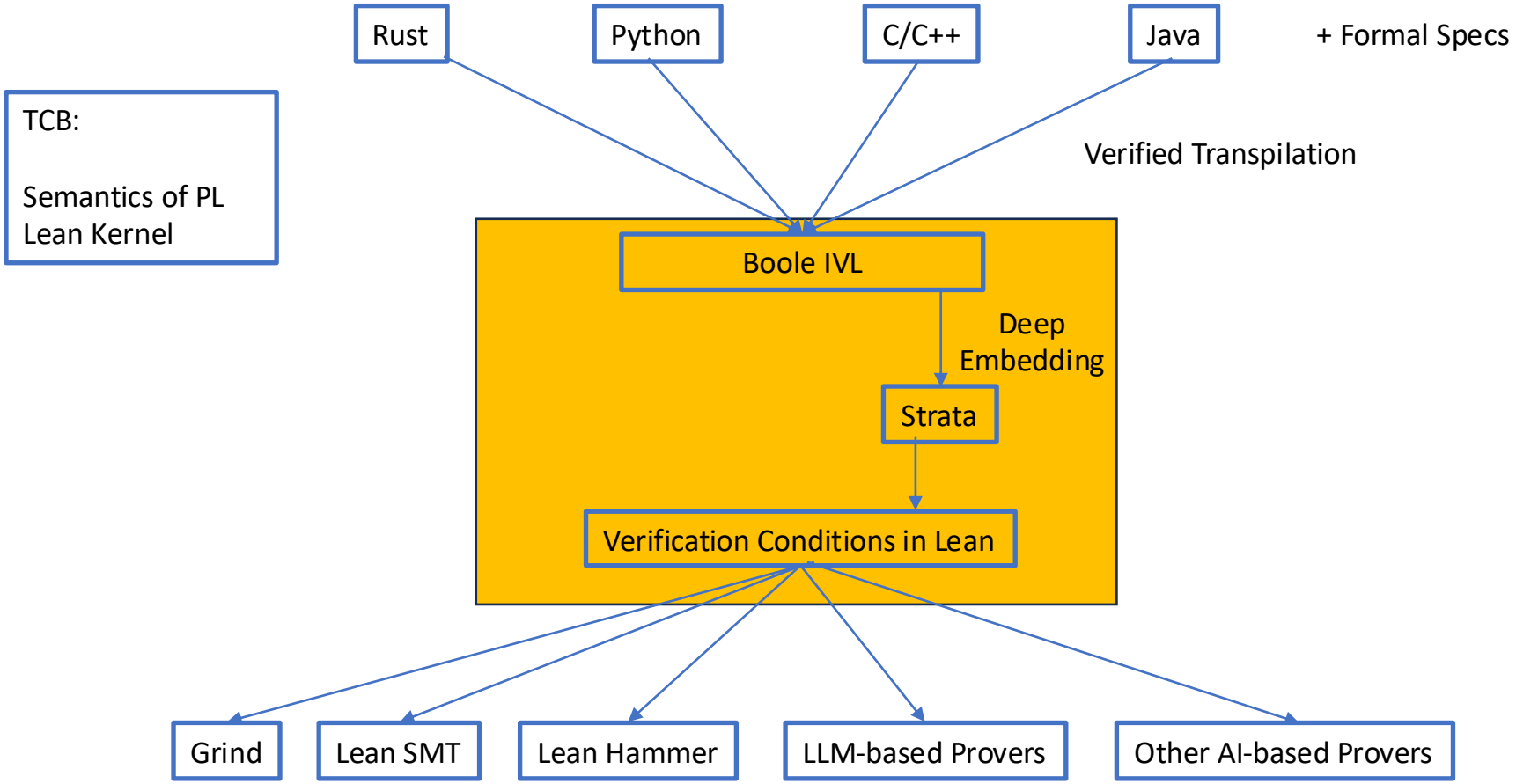
Demo



CSLIB Pillar 2 Vision



CSLIB Pillar 2 Vision



Boole Working Group

Josh Cohen

AWS

Shilpi Goel

AWS

Harun Khan

Stanford

Lydia Kondylidou

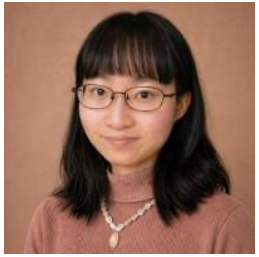
LMU Munich

Abdalrhman Mohamed

Stanford

Amitayush Thakur

UT Austin



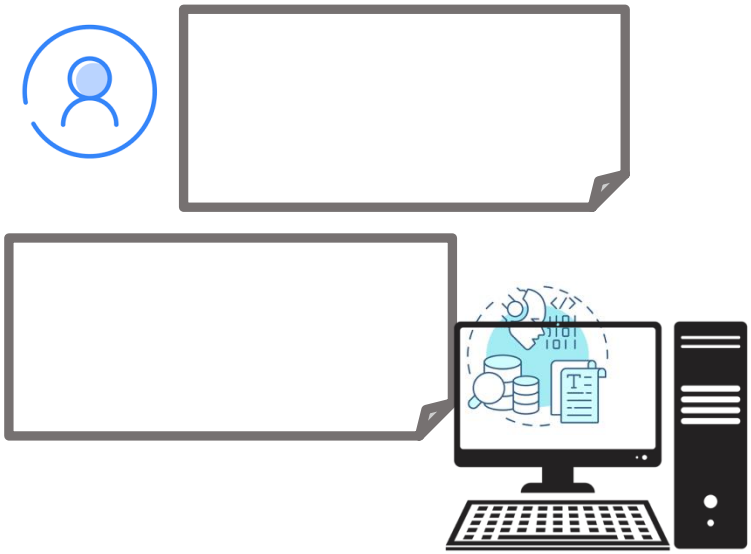
Cheng Zhang

CMU/Stanford/ UPenn

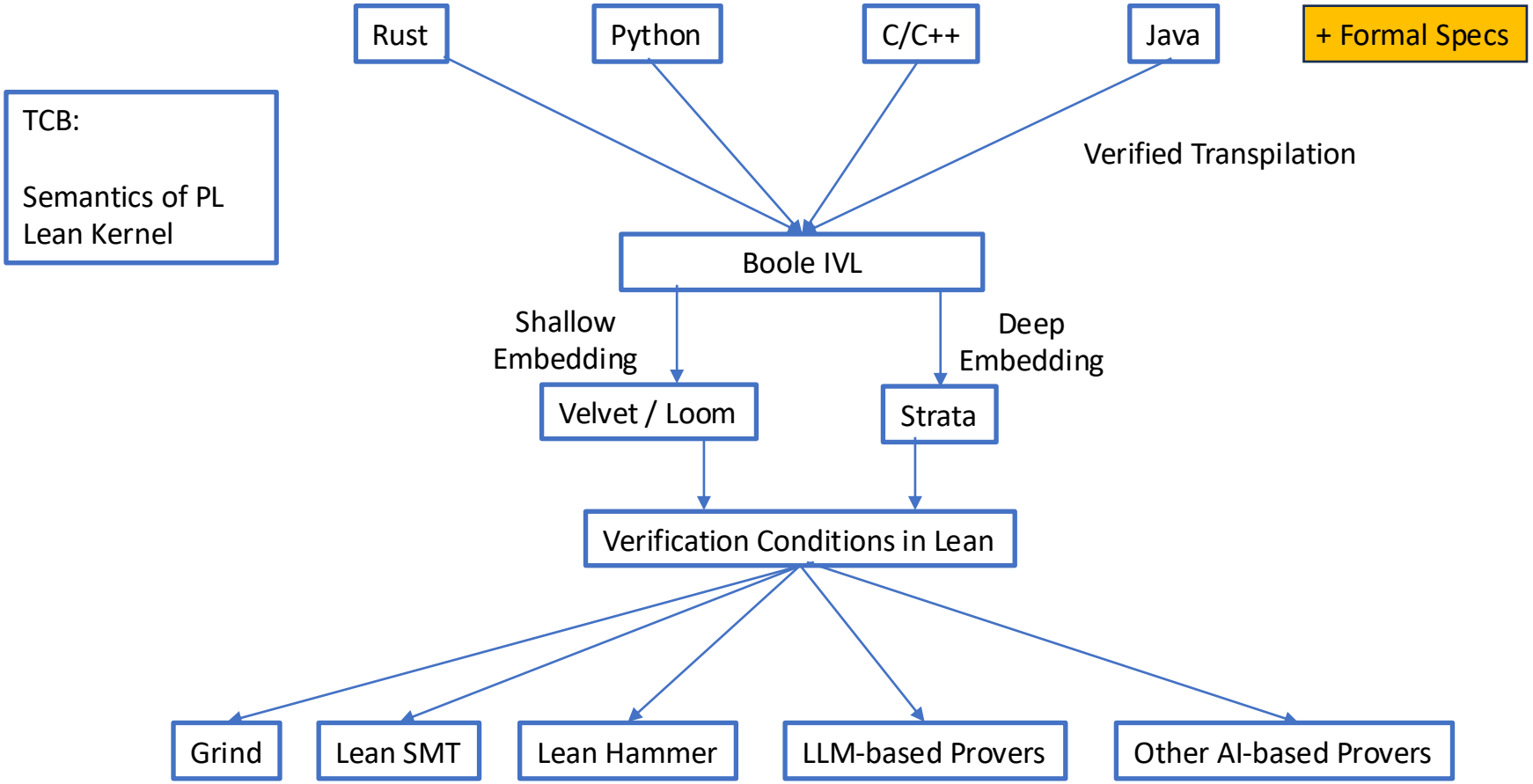
Boole Design Principles

- Boole should look like pseudocode and be easy for humans to understand
- Boole is embedded in Lean, and tools for reasoning about Boole should minimize their trusted computing base (TCB)
- Verification conditions should be generated as Lean goals and should be intuitive for humans to read and easily connectable to the code they came from

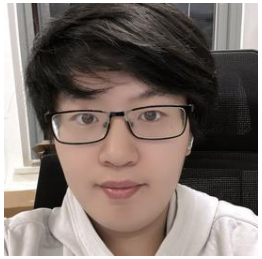
Demo



CSLIB Verification Vision



AI-Assisted Specification Generation




Chuyue (Livia) Sun
Stanford

Yican Sun
Peking University

Daneshvar Amrolahi
Stanford

Ethan Zhang
Stanford

Shuvendu Lahiri
Microsoft Research

Shan Lu
Microsoft Research

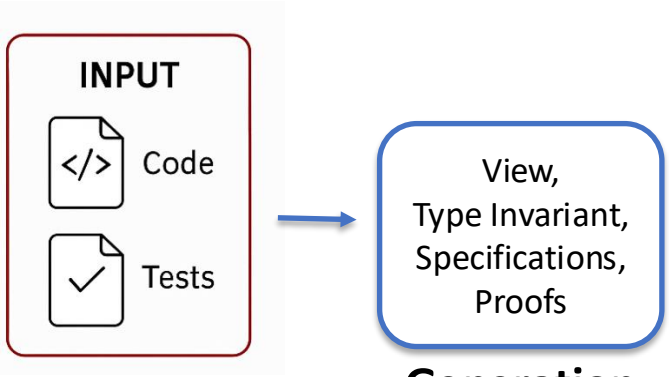
David Dill
Stanford

VeriStruct: AI-assisted Automated Verification of Data-Structure Modules in Verus by Chuyue Sun, Yican Sun, Daneshvar Amrollahi, Ethan Zhang, Shuvendu Lahiri, Shan Lu, David Dill and Clark Barrett, *TACAS 2026*

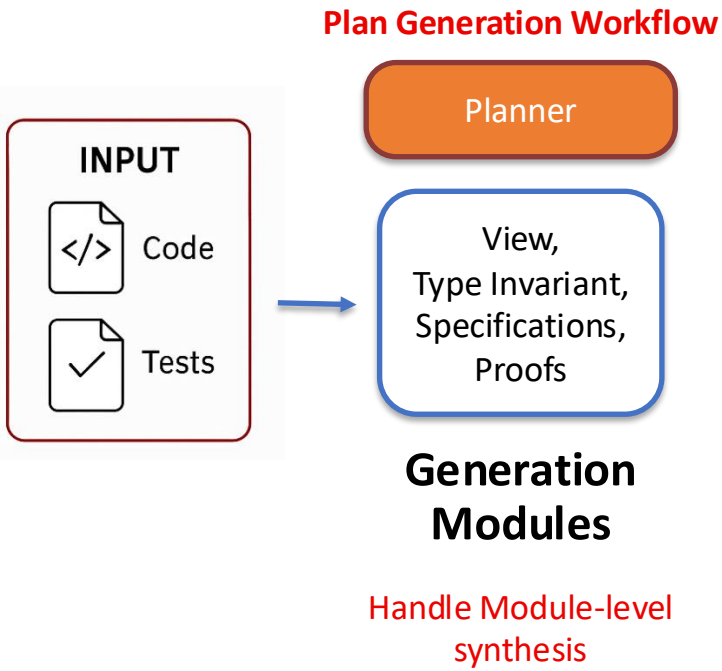
Our Framework

Tests: exercise the code and include assertions

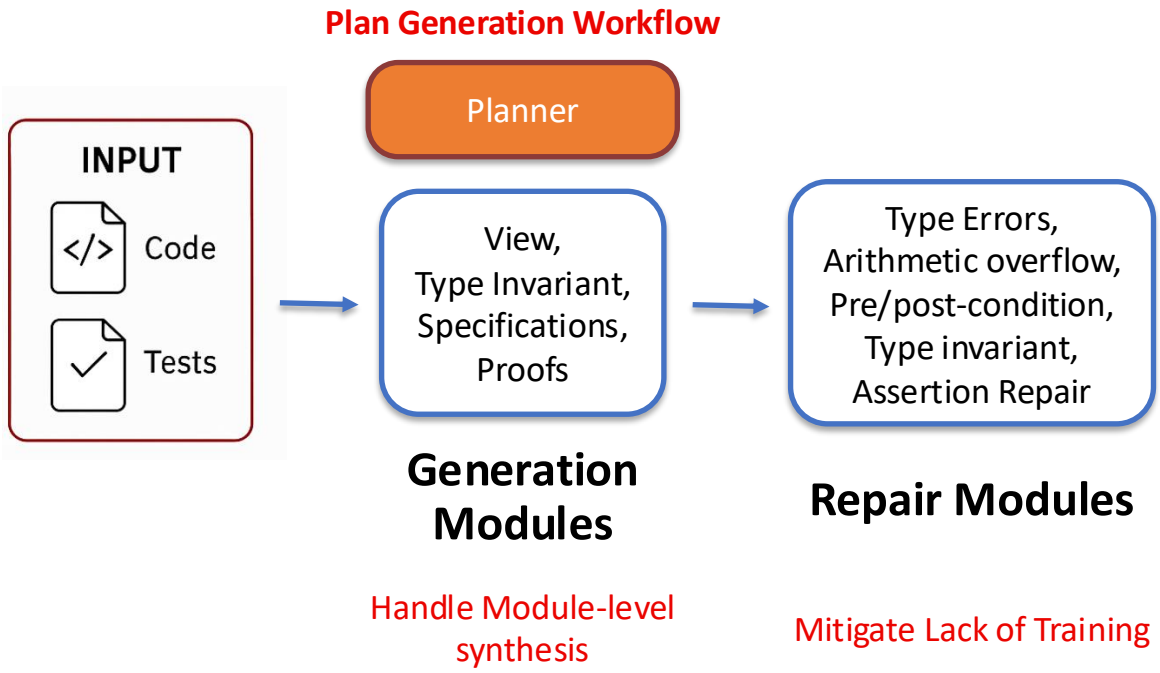
Workflow


INPUT
Code
Tests
View,
Type Invariant,
Specifications,
Proofs
Generation Modules
Handle Module-level synthesis

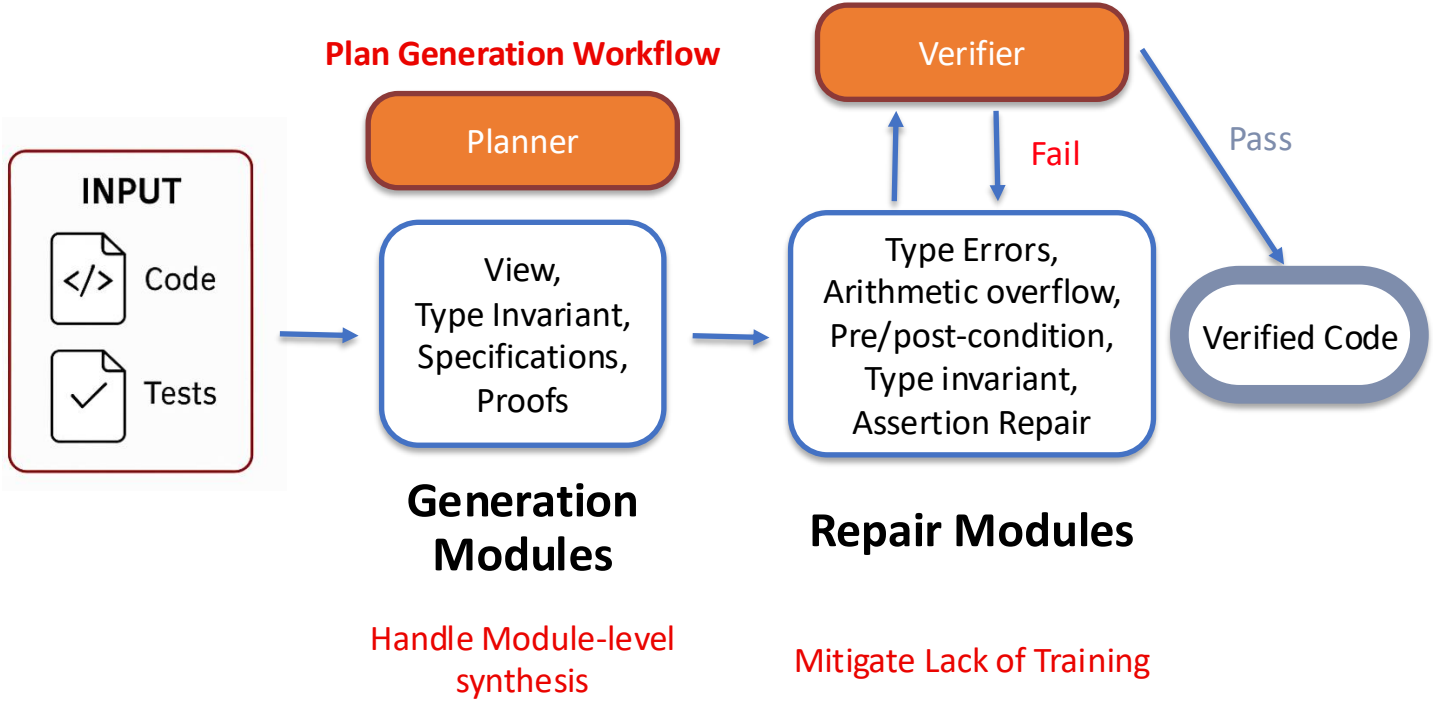
Workflow


Plan Generation Workflow
Planner
INPUT
Code
Tests
View,
Type Invariant,
Specifications,
Proofs
Generation Modules
Handle Module-level synthesis

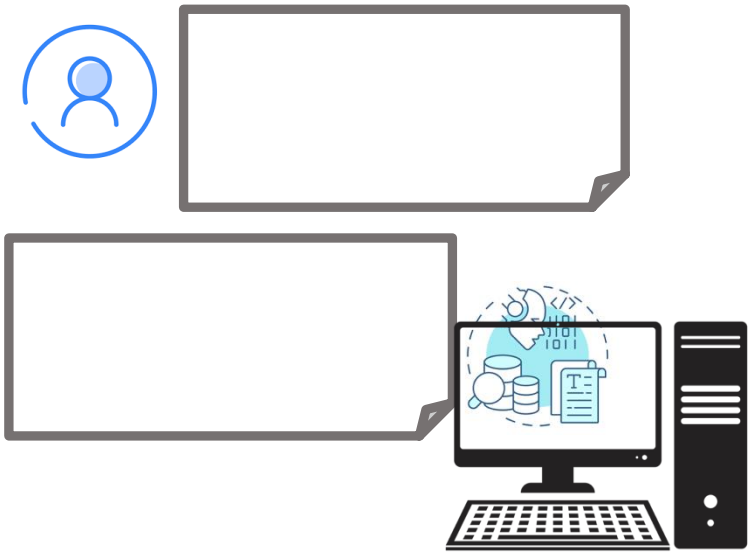
Workflow

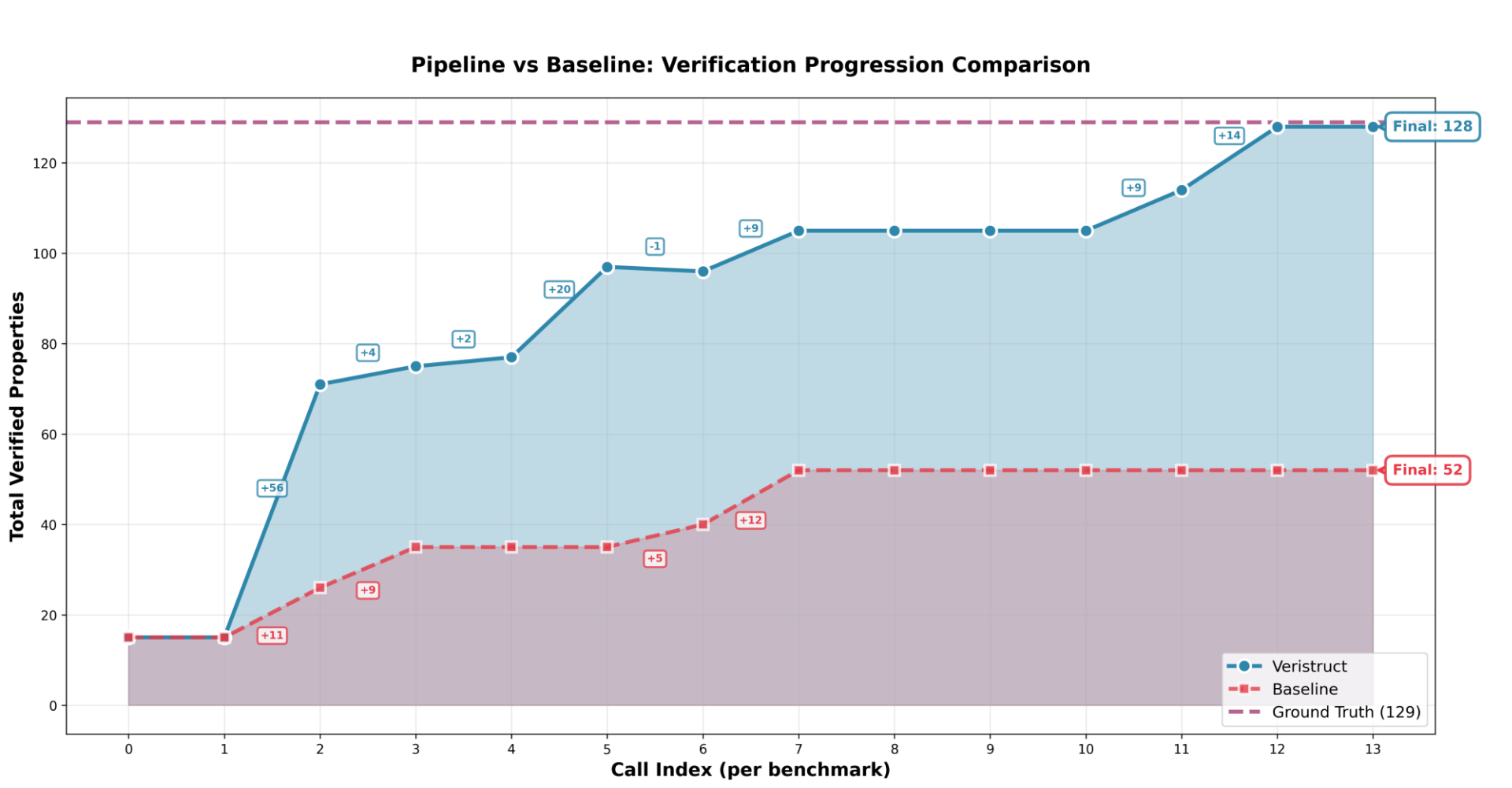

Plan Generation Workflow
Planner
INPUT
Code
Tests
View,
Type Invariant,
Specifications,
Proofs
Type Errors,
Arithmetic overflow,
Pre/post-condition,
Type invariant,
Assertion Repair
Generation Modules
Repair Modules
Handle Module-level synthesis
Mitigate Lack of Training

Workflow


Plan Generation Workflow
Verifier
Planner
Fail
Pass
INPUT
Code
Tests
View,
Type Invariant,
Specifications,
Proofs
Type Errors,
Arithmetic overflow,
Pre/post-condition,
Type invariant,
Assertion Repair
Verified Code
Generation Modules
Repair Modules
Handle Module-level synthesis
Mitigate Lack of Training

Demo




Pipeline vs Baseline: Verification Progression Comparison
Total Verified Properties
Call Index (per benchmark)
0
20
40
60
80
100
120
0
1
2
3
4
5
6
7
8
9
10
11
12
13
+56
+4
+2
+20
-1
+9
+9
+14
Final: 128
+11
+9
+5
+12
Final: 52
Veristruct
Baseline
Ground Truth (129)

Next Steps

- Larger, more complex programs
- Programs in realistic programming languages
- More work on infrastructure: Boole / Strata / VC generation
- Better automation for theorem proving in Lean
  - Improve automated reasoning-based algorithms (grind, Lean SMT)
  - Specialized RL and LLM-based agents (Tengyu and startups)
- Build up a library of verified programs in CSLib

How to get involved

- Visit the website:
  - cslib.io
- Read the CSLib whitepaper:
  - https://arxiv.org/abs/2602.04846
- Join the Zulip channels:
  - #CSLib
  - #CSLib: Code Reasoning
- Visit CSLib on GitHub
  - CONTRIBUTING.md

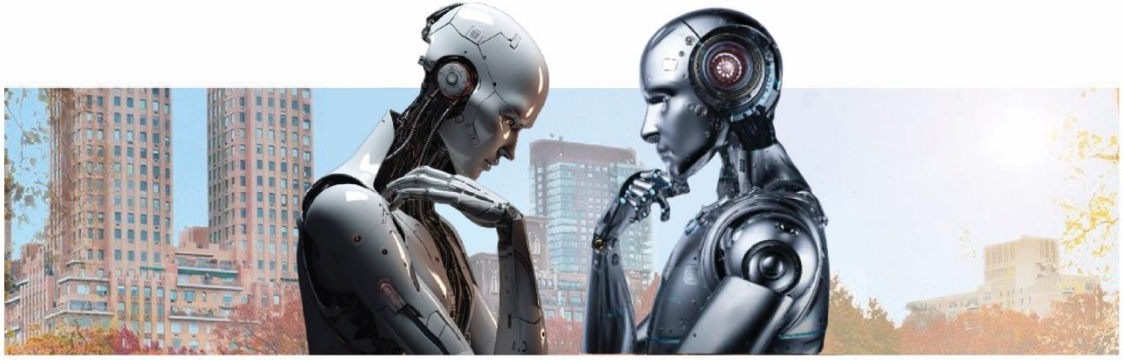
Code reasoning and AI Safety

- Code reasoning is “base camp” for the AI Safety Everest

- General capability to reason about code unlocks many avenues that are currently unavailable
  - “Proof-carrying code” could establish new mechanisms for trust
  - Agents can self-organize by providing guarantees to each other

- AI-assisted code verification would be one of the first examples of AI *containment* – i.e., even a superintelligent AI would be bound by the formal framework

WHEN
AR
MET
AI



Questions?

SAFE

Stanford | Center for AI Safety

CENTAUR

Stanford | Center for Automated Reasoning