Safety by Shared Synthesis

•••

Berkeley Seminar 20240924

Shaowei Lin

Joint work with Atlas Computing

TOPOS INSTITUTE

Atlas Computing

Liveness and Safety of Sociotechnical Systems

Intentional attacks/ accidental mistakes

Human-led/ AI-enabled

Do we want to fix our sociotechnical systems one bug at a time?

CrowdStrike and Microsoft: What we know about global IT outage

19 July 2024

Robert Plummer
BBC News

Tom Gerken
Technology reporter

There were long lines at Barcelona airport, as passengers waited to be checked in manually

https://www.bbc.com/news/articles/cp4wnrxqlewo

FEBRUARY 26, 2024

Press Release: Future Software Should Be Memory Safe

ONCD › BRIEFING ROOM › PRESS RELEASE

Leaders in Industry Support White House Call to Address Root Cause of Many of the Worst Cyber Attacks

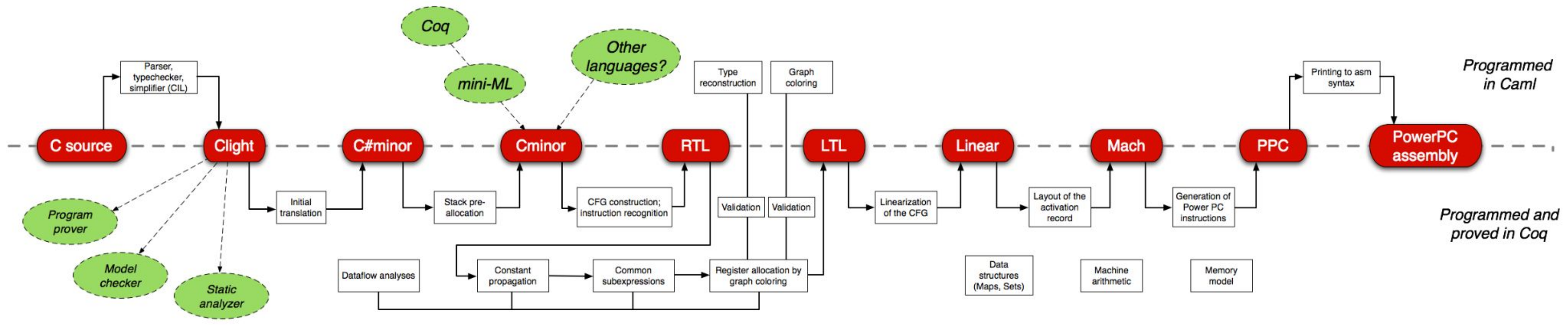
https://www.whitehouse.gov/oncd/briefing-room/2024/02/26/press-release-technical-report/

Hardening Systems with Formal Verification

"... **CompCert** is the only compiler we have tested for which Csmith cannot find wrong-code errors. This is not for lack of trying: we have devoted about six CPU-years to the task."

Yang, X., Chen, Y., Eide, E. and Regehr, J., 2011, June. Finding and understanding bugs in C compilers. In Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation (pp. 283-294).

But writing formal specifications, implementations and proofs is hard!



Strategy

Strategy 1: Scaling FV with AI Assistance

Large language models (LLMs) are becoming good at generating code.

How do we ensure they produce code that is correct?

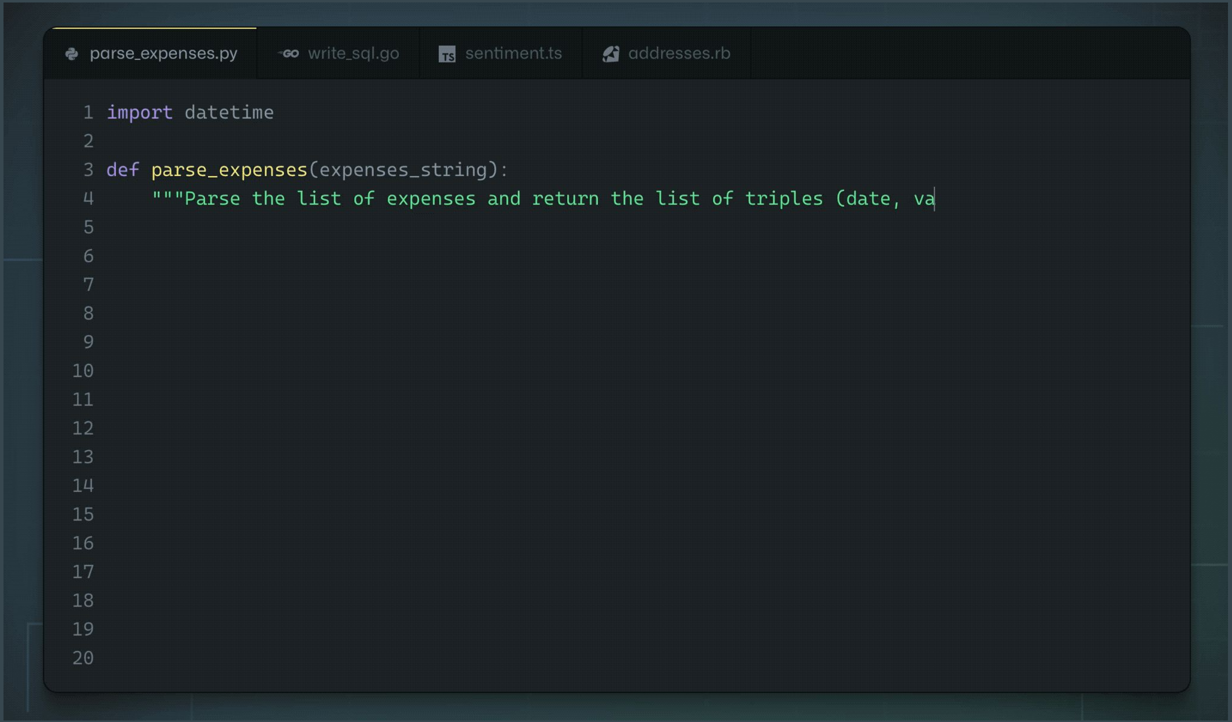
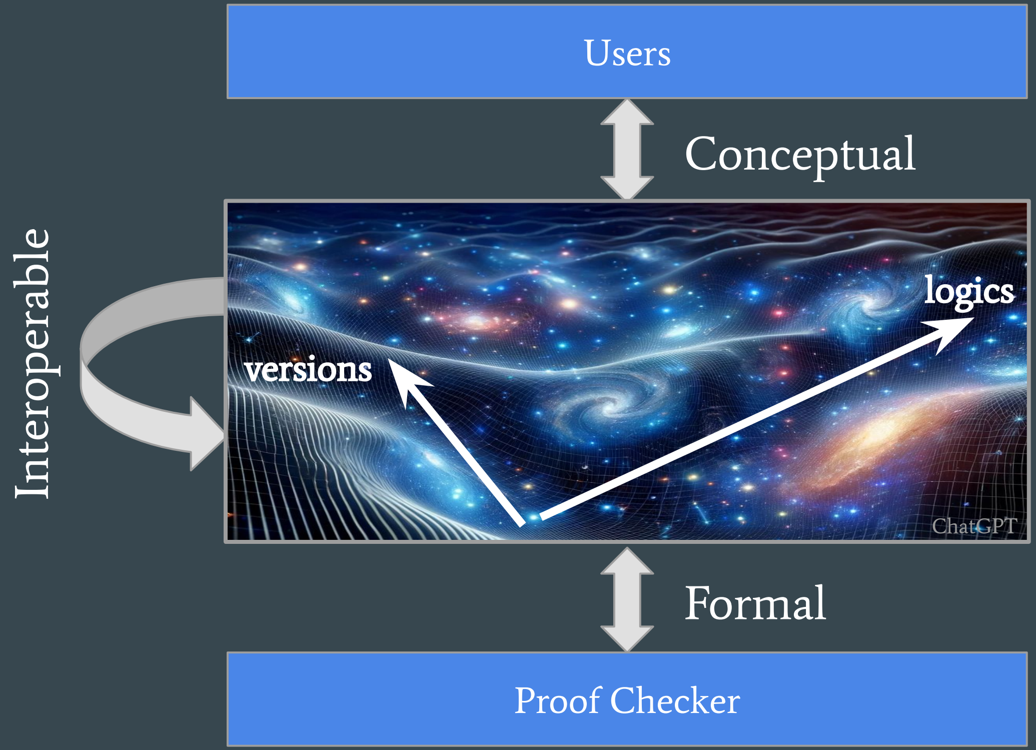


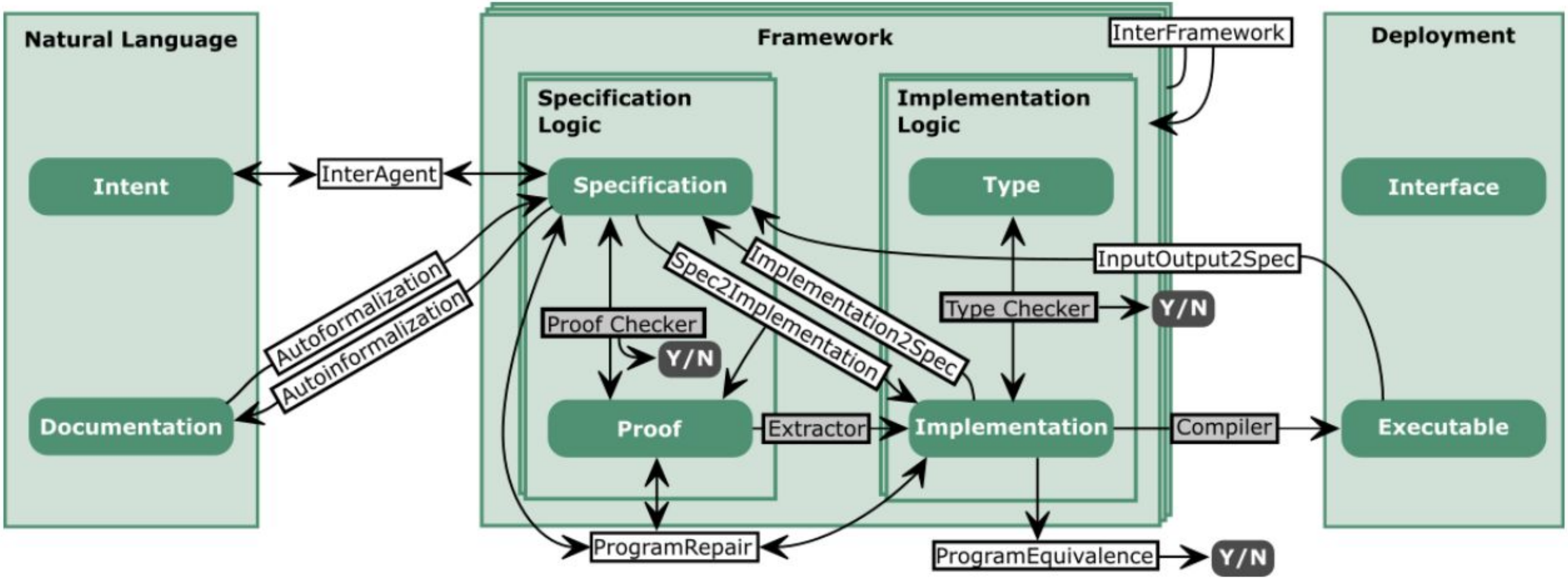
Image Credits: GitHub Copilot

Strategy 2: Making Sense Collectively with Formal Tools

Domain-specific logics (DSLs)

- Conceptual
- Interoperable
  - Between models
  - Between versions
- Formal





Toolchain

https://atlascomputing.org/ai-assisted-fv-toolchain.pdf

Joint work with Atlas Computing; See also ARIA's Safeguarded AI programme.

AI-Assistance

Translating Between Logics

e.g. C to Rust,
Coq to Lean

ChatGPT

To translate the provided Coq code into the Lean programming language, we need to understand the structure of the Coq code and then replicate that logic in Lean. The Coq code defines a function `instrDenote` which takes an instruction `i` and a stack `s`, and then processes the instruction to potentially modify the stack.

Here's how the same functionality can be implemented in Lean using a similar pattern:

lean Copy code

```lean
inductive instr
| iConst : ℕ → instr
| iBinop : (ℕ → ℕ → ℕ) → instr

open instr

def binopDenote (b : ℕ → ℕ → ℕ) (arg1 arg2 : ℕ) : ℕ :=
  b arg1 arg2

def instrDenote : instr → list ℕ → option (list ℕ)
| (iConst n) s := some (n :: s)
| (iBinop b) (arg1 :: arg2 :: s) := some (binopDenote b arg1 arg2 :: s)
| (iBinop _) _ := none
```

You

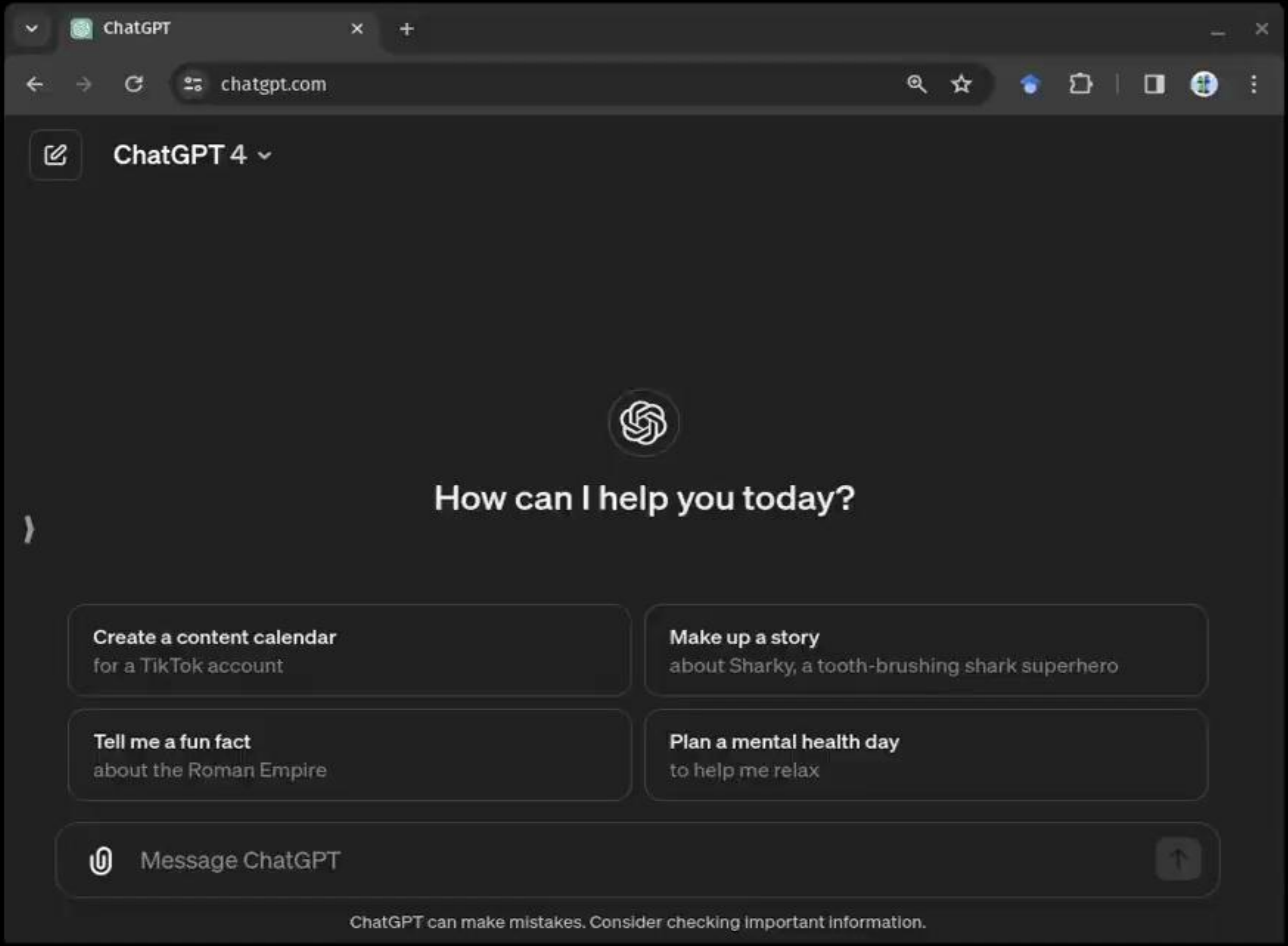
Translate the following code from the Coq programming language into the Lean programming language.

```coq
Definition instrDenote (i : instr) (s : stack) : option stack :=
 match i with
  | iConst n => Some (n :: s)
  | iBinop b =>
   match s with
    | arg1 :: arg2 :: s' => Some ((binopDenote b) arg1 arg2 :: s')
    | _ => None
   end
 end.
```

From Natural Language To Specs

Importance of legacy docs

Generating well-formed terms of DSLs

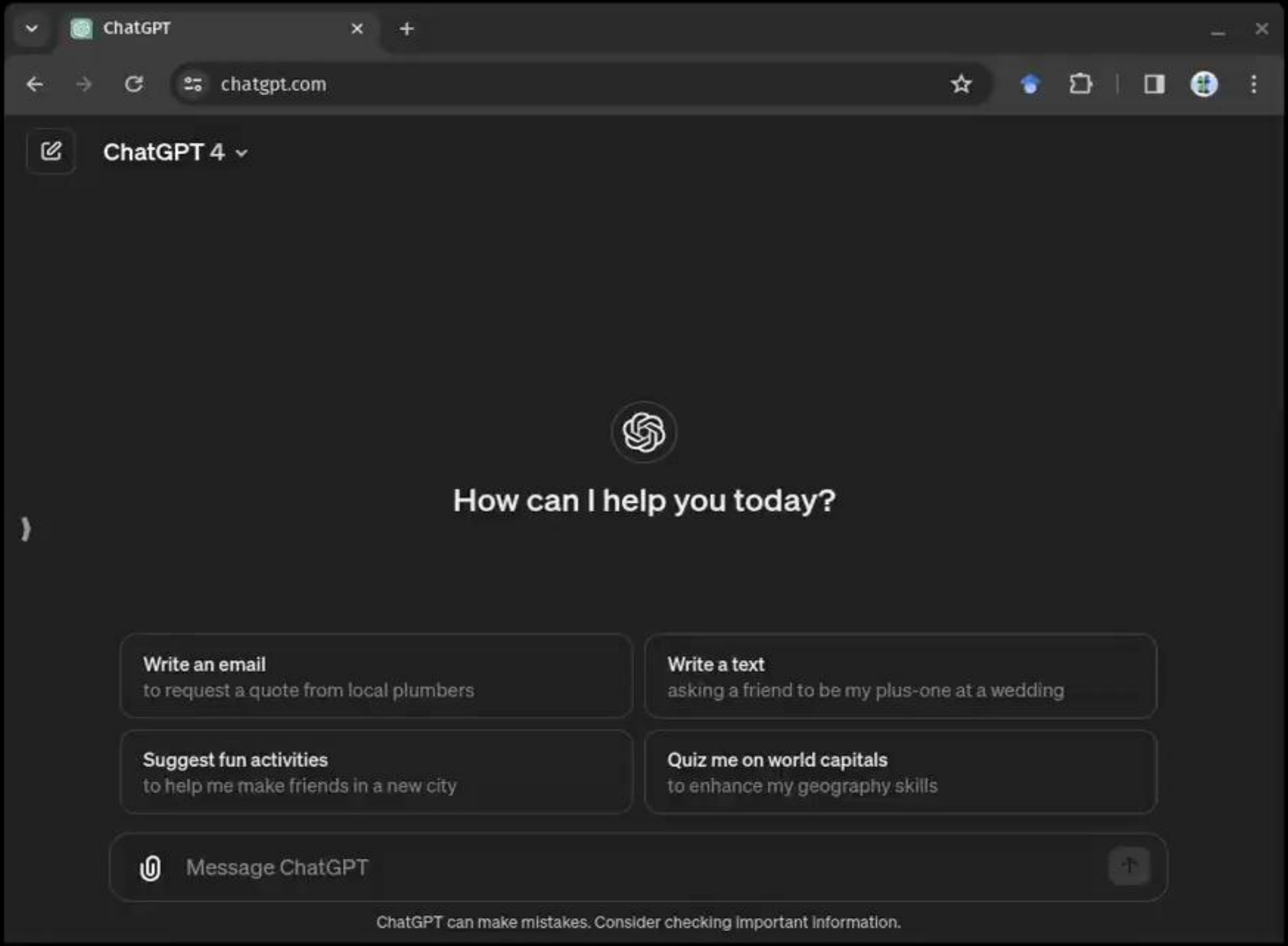


Brandfonbrener, D., Raja, S., Prasad, T., Loughridge, C., Yang, J., Henniger, S., Byrd, W.E., Zinkov, R. and Amin, N., 2024. Verified Multi-Step Synthesis using Large Language Models and Monte Carlo Tree Search. arXiv preprint arXiv:2402.08147

AI-Assistance

From Code To Specs

Importance of legacy codes



Sun, C., Sheng, Y., Padon, O. and Barrett, C., 2023. Clover: Closed-Loop Verifiable Code Generation. arXiv preprint arXiv:2310.17807.

AI-Assistance

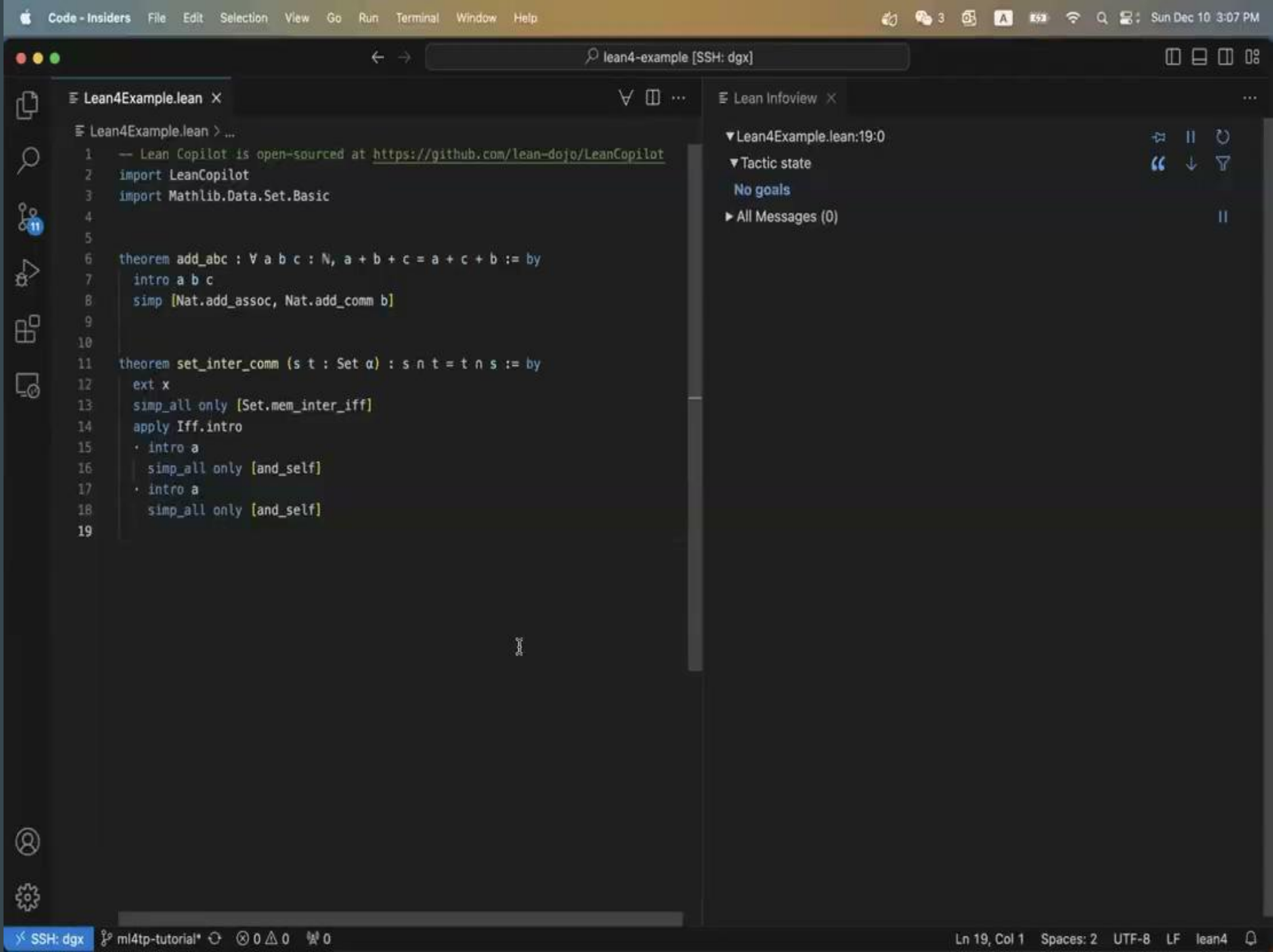
From
Specs
To Proofs

Lean Copilot

suggest_tactics

search_proof

select_premises



Synthesis

```
method maxArray(a: array<int>) returns (m: int)
    requires a != null && a.Length > 0
    ensures forall k :: 0 <= k < a.Length ==> a[k] <=
m
    ensures exists k :: 0 <= k < a.Length && a[k] ==
m
{
    m := a[0];
    var i := 1;
    while (i < a.Length)
        invariant 1 <= i <= a.Length
        invariant forall j :: 0 <= j < i ==> a[j] <=
m
        invariant exists j :: 0 <= j < i && a[j] == m
    {
        m := if m > a[i] then m else a[i];
        i := i + 1;
    }
}
```

Auto-Active Frameworks

Hoare logic, Separation logic

Automated Theorem Provers (ATP), e.g. Dafny, Frama-C, Verus

Expressive Frameworks

Dependent type theory

Interactive Theorem Provers (ITP), e.g. Coq, Lean, Agda

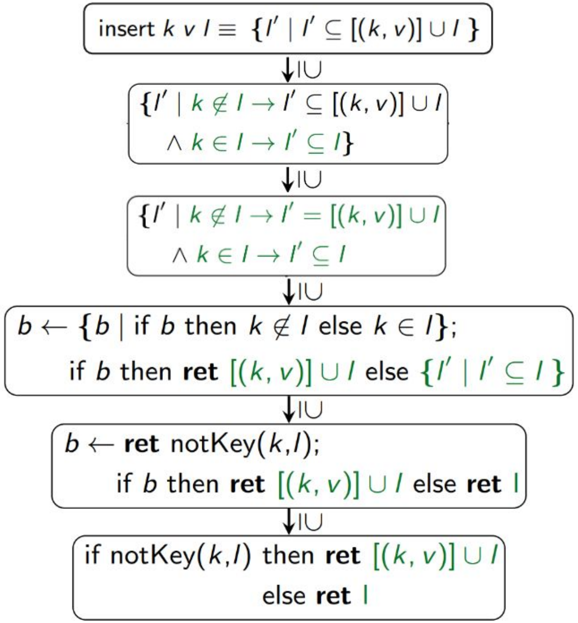
```
-- Import necessary libraries for proving
import data.nat.basic
import tactic

-- Using the previously define.1d factorial function
def factorial : ℕ → ℕ
| 0     := 1
| (n+1) := (n + 1) * factorial n

-- Proof that factorial is always positive
theorem factorial_pos :  ∀ n : ℕ, factorial n > 0 :=
begin
  -- Apply mathematical induction on n
  apply nat.rec,
  -- Base case: show that factorial 0 is 1, which is positive
  { show factorial 0 > 0,
    rw factorial,  -- factorial 0 = 1 by definition
    exact nat.zero_lt_one, },
  -- Inductive step: assume factorial n is positive,
  -- prove factorial (n+1) is positive
  { intros n h,
    show factorial (n + 1) > 0,
    rw factorial,  -- expand factorial (n + 1)
    apply nat.mul_pos,  -- product of positives is positive
    show n + 1 > 0,
    exact nat.succ_pos n,
    exact h, }
end
```

Insight 1: Tactic-By-Tactic, Not Token-By-Token!

Refinement-based synthesis -
functionality without sacrificing *performance*
e.g. fiat-crypto used by >95% of HTTPS links



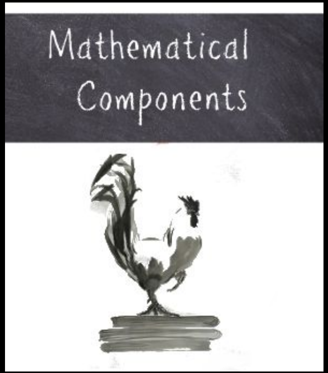

```
hone method insert. {
        StartMethod.
        setoid_rewrite refine_ReplaceUsedKeyAdd.
        setoid_rewrite refine_SubEnsembleInsert.
        autorewrite with monad laws.
        setoid_rewrite refine_pick_KeyToBeReplaced min.
        setoid_rewrite refine_If_Then_Else_Bind.
        autorewrite with monad laws.
        setoid_rewrite refine_If_Opt_Then_Else_Bind.
        autorewrite with monad laws.
        setoid_rewrite refine_pick_CacheADTwLogIndex AbsR.
        setoid_rewrite refine_pick_KVEnsembleInsertRemove
                              with (1 := EquivKeys H).
        setoid_rewrite refine_pick_KVEnsembleInsert
                              with (1 := EquivKeys H).
        autorewrite with monad laws; simpl.
        finish honing. }
```

Insight 2: Let Machines Fill In The Blanks!

Categorize goals into **typeclasses** and design generic tactics that infer missing class parameters.

e.g. proof of Four Color Thm and Odd Order Thm with mathematical components

With generic tactics, we can train AI models that generalize well across different domains.




```
Definition eq_axiom T (e : rel T) := forall x y, reflect (x = y) (e x y).

HB.mixin Record hasDecEq T := { eq_op : rel T; eqP : eq_axiom eq_op }.

#[mathcomp(axiom="eq_axiom"), short(type="eqType")]
HB.structure Definition Equality := { T of hasDecEq T }.
```

```
Lemma eq_refl (T : eqType) (x : T) : x == x. Proof. exact/eqP. Qed.
Notation eqxx := eq_refl.

Lemma eq_sym (T : eqType) (x y : T) : (x == y) = (y == x).
Proof. exact/eqP/eqP. Qed.

#[global] Hint Resolve eq_refl eq_sym : core.
```

Large-Scale Collaborative Math Theorem Proving

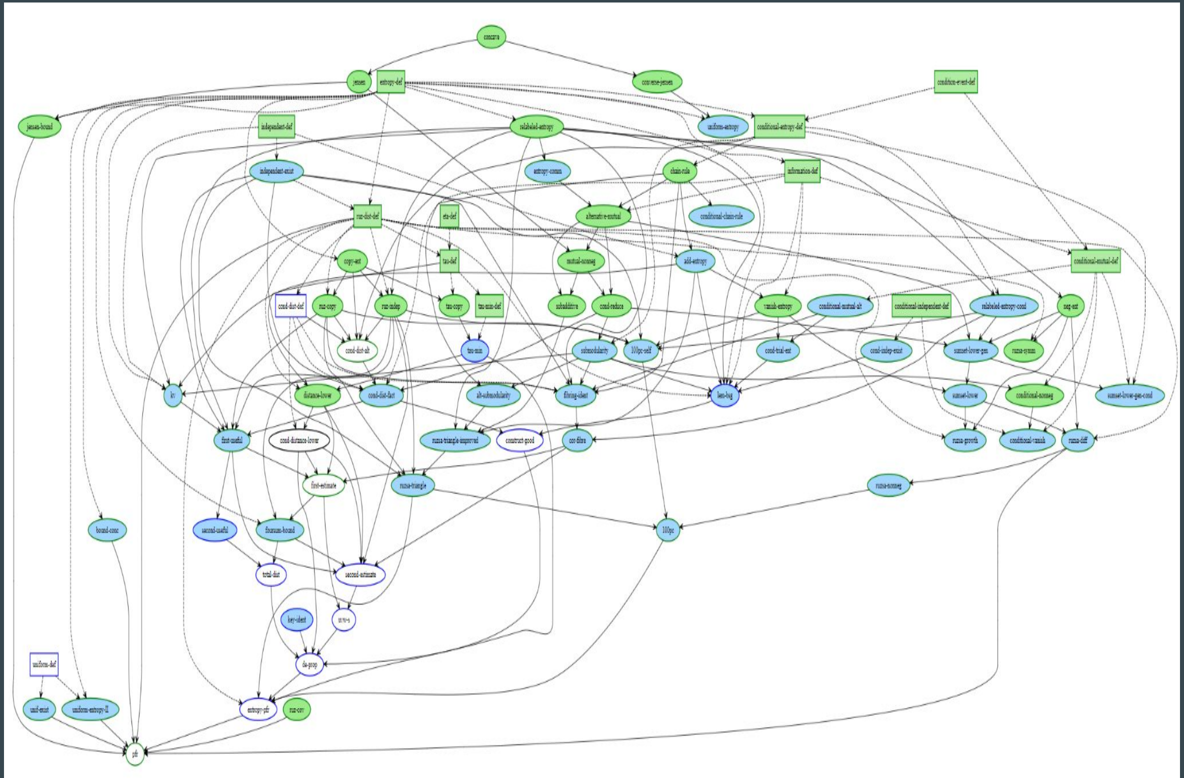
Shared Synthesis
= Blueprint + Lean
= ...
= Profit!!

Liquid Tensor Experiment

PFR Conjecture

∞-cosmos Formalization

National Academies report on AI-assisted math reasoning

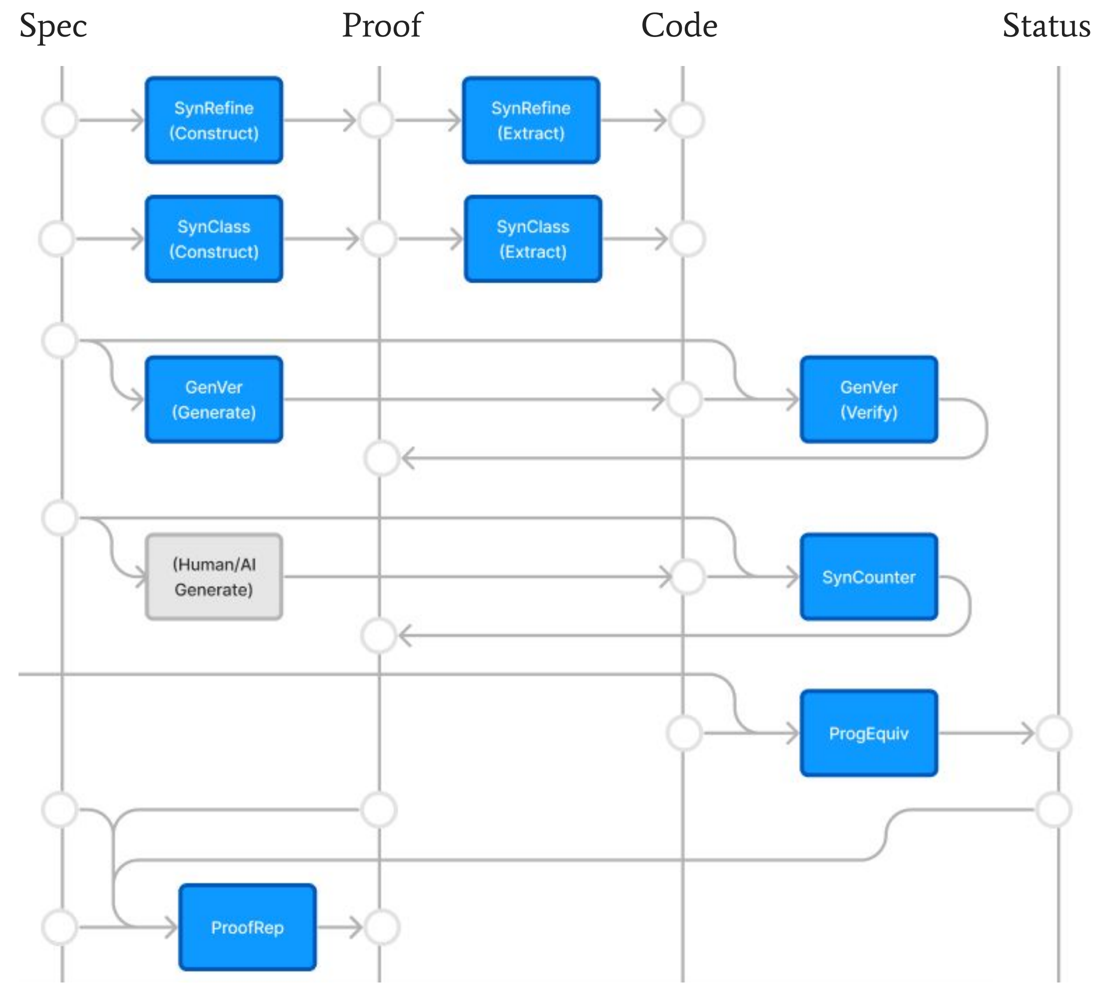


Program Equivalence

- Check if two implementations have the same behavior (ProgEquiv)
- Otherwise give a counterexample (ProgCounter)

Proof Repair

- Fix proofs when libraries, specs, implementations change (ProofRep)



Safety by Shared Synthesis

Strategy	Scaling formal verification with AI assistance
Strategy	Making sense collectively with formal tools
AI-Assistance	Logics, specs, codes, proofs
Synthesis	Tactic-by-tactic, not token-by-token
Synthesis	Let machines fill in the blanks

Thank you!

https://atlascomputing.org/ai-assisted-fv-toolchain.pdf

Lessons

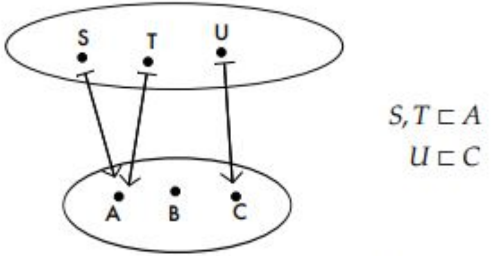
Functors are Type Refinement Systems

Now, for the remainder of the section we will assume a fixed, arbitrary functor $\mathbf{U} : \mathcal{D} \to \mathcal{T}$, and consider various notions relative to $\mathbf{U}$.

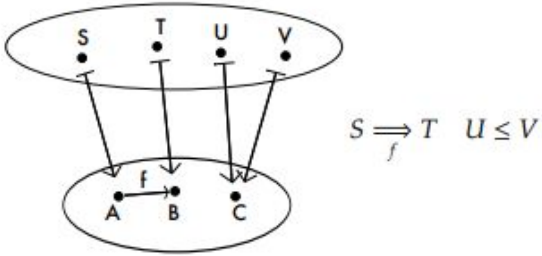
Definition 3. We say that an object $S \in \mathcal{D}$ **refines** an object $A \in \mathcal{T}$ if $\mathbf{U}(S) = A$.

Definition 4. A **typing judgment** is a triple (S, f, T) such that S and T refine the domain and codomain of f, respectively, i.e., such that $f : A \to B$, $\mathbf{U}(S) = A$ and $\mathbf{U}(T) = B$, for some arbitrary A and B. In the special case where $f = \mathrm{id}$ (implying that $\mathbf{U}(S) = \mathbf{U}(T)$), we also call this a **subtyping judgment**.

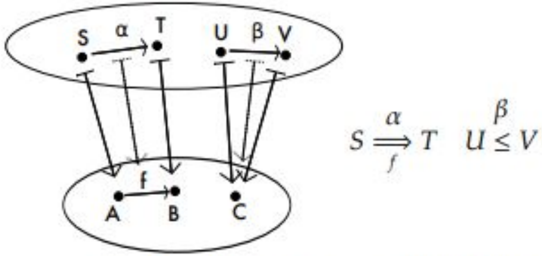
Definition 5. A **derivation** of a typing judgment (S, f, T) is a morphism $\alpha : S \to T$ in $\mathcal{D}$ such that $\mathbf{U}(\alpha) = f$. ■



(a) Type refinement



(b) Typing and subtyping judgments



(c) Derivations of typing and subtyping judgments

https://www.irif.fr/~mellies/papers/functors-are-type-refinement-systems.pdf

Category of Derivations

$$\frac{\Gamma \vdash f : A \to B \quad \Gamma \vdash a : A}{\Gamma \vdash f(a) : B} \quad \text{Elimination}$$

$^{\times}$-form: $A, B \rightsquigarrow A \times B$	$^{\to}$-form: A and $B \rightsquigarrow A \to B$
$^{\times}$-intro: $a : A, b : B \rightsquigarrow (a, b) : A \times B$	$^{\to}$-intro: $x : A \vdash b : B \rightsquigarrow \lambda x.b : A \to B$
$^{\times}$-elim: $p : A \times B \rightsquigarrow pr_1 p : A, pr_2 p : B$	$^{\to}$-elim: $f : A \to B, a : A \rightsquigarrow f(a) : B$

Category of Terms

Objects often same as those the category of derivations but without proof information

e.g. $\mathbb{N}$ vs $\{n : \mathbb{N} \mid \exists\ k : \mathbb{N}, 2k = n\}$

Hoare Logic involves Refinement

Example 4. The general approach of Hoare logic [12] provides a natural class of examples of refinement systems, to a first approximation defined as follows (we will consider a more nuanced view in Section 5):

- Take $\mathcal{T}$ as a category with one object W corresponding to the state space, and with morphisms $c : W \to W$ corresponding to program commands, identified with state transformers.
- Take $\mathcal{D}$ as a category whose objects are predicates ϕ over states, and whose morphisms $\phi \to \psi$ are pairs of a state transformer c together with a verification that c takes any state satisfying ϕ to a state satisfying ψ.
- Let $\mathbf{U} : \mathcal{D} \to \mathcal{T}$ be the evident forgetful functor, mapping every ϕ to W and every verification about c to c itself.

$$\frac{\{P\}S\{Q\} \quad , \quad \{Q\}T\{R\}}{\{P\}S;T\{R\}}$$

composition rule

https://www.irif.fr/~mellies/papers/functors-are-type-refinement-systems.pdf

Separation Logic involves Refinement

“Separation logic assertions describe "states" consisting of a store and a heap, roughly corresponding to the state of local (or stack-allocated) variables and dynamically-allocated objects in common programming languages such as C and Java.

A store is a function mapping variables to values.

A heap is a partial function mapping memory addresses to values.”

$$\frac{\{P\}\ C\ \{Q\}}{\{P * R\}\ C\ \{Q * R\}}$$

frame rule

Rust is a formal verification framework!

Ownership conditions are checked at compile time, not at run time.

```rust
#[allow(dead_code)]
#[derive(Clone, Copy)]
struct Book {
    // `&'static str` is a reference to a string allocated in read only memory
    author: &'static str,
    title: &'static str,
    year: u32,
}

// This function takes a reference to a book
fn borrow_book(book: &Book) {
    println!("I immutably borrowed {} - {} edition", book.title, book.year);
}

// This function takes a reference to a mutable book and changes `year` to 2014
fn new_edition(book: &mut Book) {
    book.year = 2014;
    println!("I mutably borrowed {} - {} edition", book.title, book.year);
}

fn main() {
    // Create an immutable Book named `immutabook`
    let immutabook = Book {
        // string literals have type `&'static str`
        author: "Douglas Hofstadter",
        title: "Gödel, Escher, Bach",
        year: 1979,
    };

    // Create a mutable copy of `immutabook` and call it `mutabook`
    let mut mutabook = immutabook;

    // Immutably borrow an immutable object
    borrow_book(&immutabook);

    // Immutably borrow a mutable object
    borrow_book(&mutabook);

    // Borrow a mutable object as mutable
    new_edition(&mut mutabook);

    // Error! Cannot borrow an immutable object as mutable
    new_edition(&mut immutabook);
    // FIXME ^ Comment out this line
}
```

Separation logic minimally assumes memory safety; Rust takes care of that!

In verifying Rust code, you can focus on its functional properties.

The Aeneas verification framework was designed with this principle in mind.

separating resources

The turn of the millennium marked the beginning of a new era with the introduction of separation by O'Hearn and Reynolds[2] logic which builds upon the notions of *logical resources* first introduced by Girard for linear logic. In separation logic, we treat ownership of memory as a non-duplicable resource , if I have ownership of some address `a`, I can't also give you ownership of `a`, that would be like if I could turn one dollar into two. To combine resources we have a *separating conjunction* `a &*& b` which represents the combination of disjoint resources `a` and `b` (like two separate addresses).

https://xav.io/blog/rust-formal-verification/
https://github.com/AeneasVerif/aeneas

Synthesis involves Refinement

$$\text{insert } k\ v\ l \equiv \{l' \mid l' \subseteq [(k,v)] \cup l\}$$

$$\big\downarrow \supseteq$$

$$\{l' \mid k \notin l \to l' \subseteq [(k,v)] \cup l \;\wedge\; k \in l \to l' \subseteq l\}$$

$$\big\downarrow \supseteq$$

$$\{l' \mid k \notin l \to l' = [(k,v)] \cup l \;\wedge\; k \in l \to l' \subseteq l$$

$$\big\downarrow \supseteq$$

$$b \leftarrow \{b \mid \text{if } b \text{ then } k \notin l \text{ else } k \in l\}; \quad \text{if } b \text{ then } \mathbf{ret}\ [(k,v)] \cup l \text{ else } \{l' \mid l' \subseteq l\}$$

$$\big\downarrow \supseteq$$

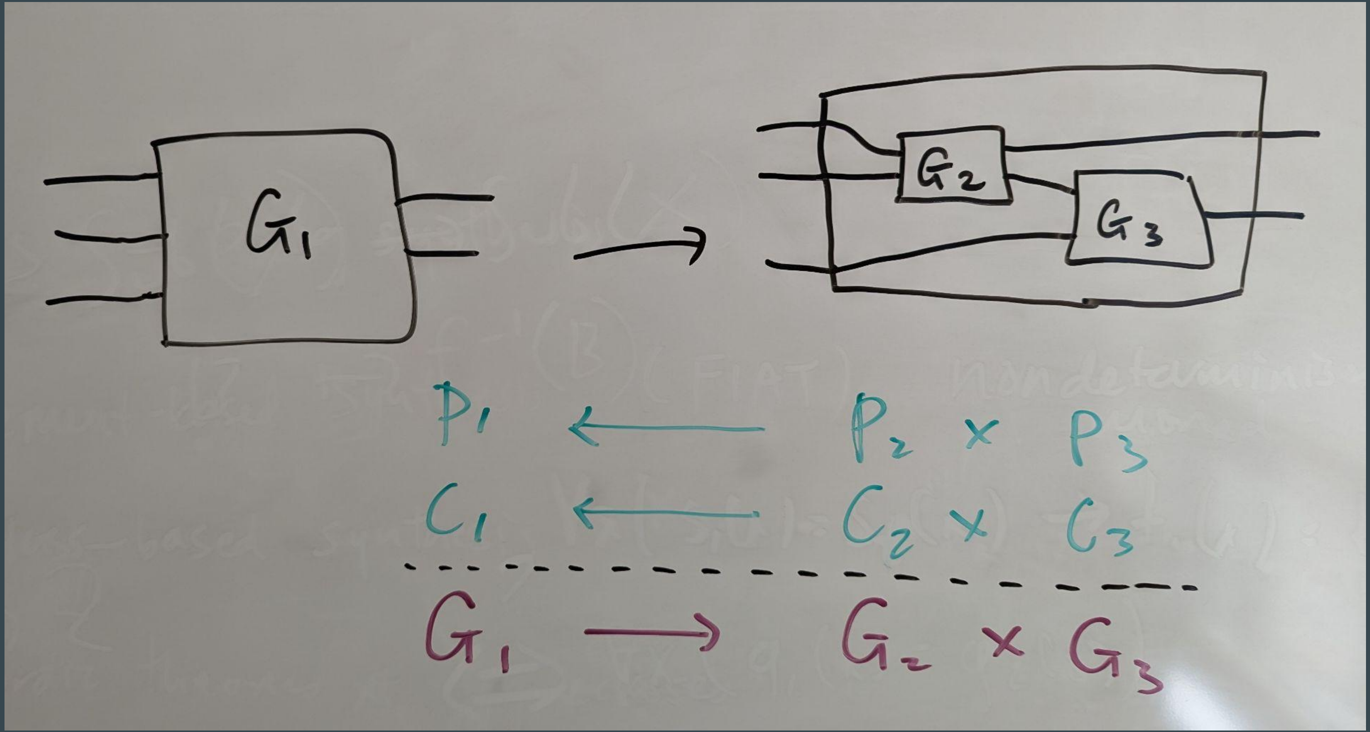
$$b \leftarrow \mathbf{ret}\ \text{notKey}(k,l); \quad \text{if } b \text{ then } \mathbf{ret}\ [(k,v)] \cup l \text{ else } \mathbf{ret}\ l$$

$$\big\downarrow \supseteq$$

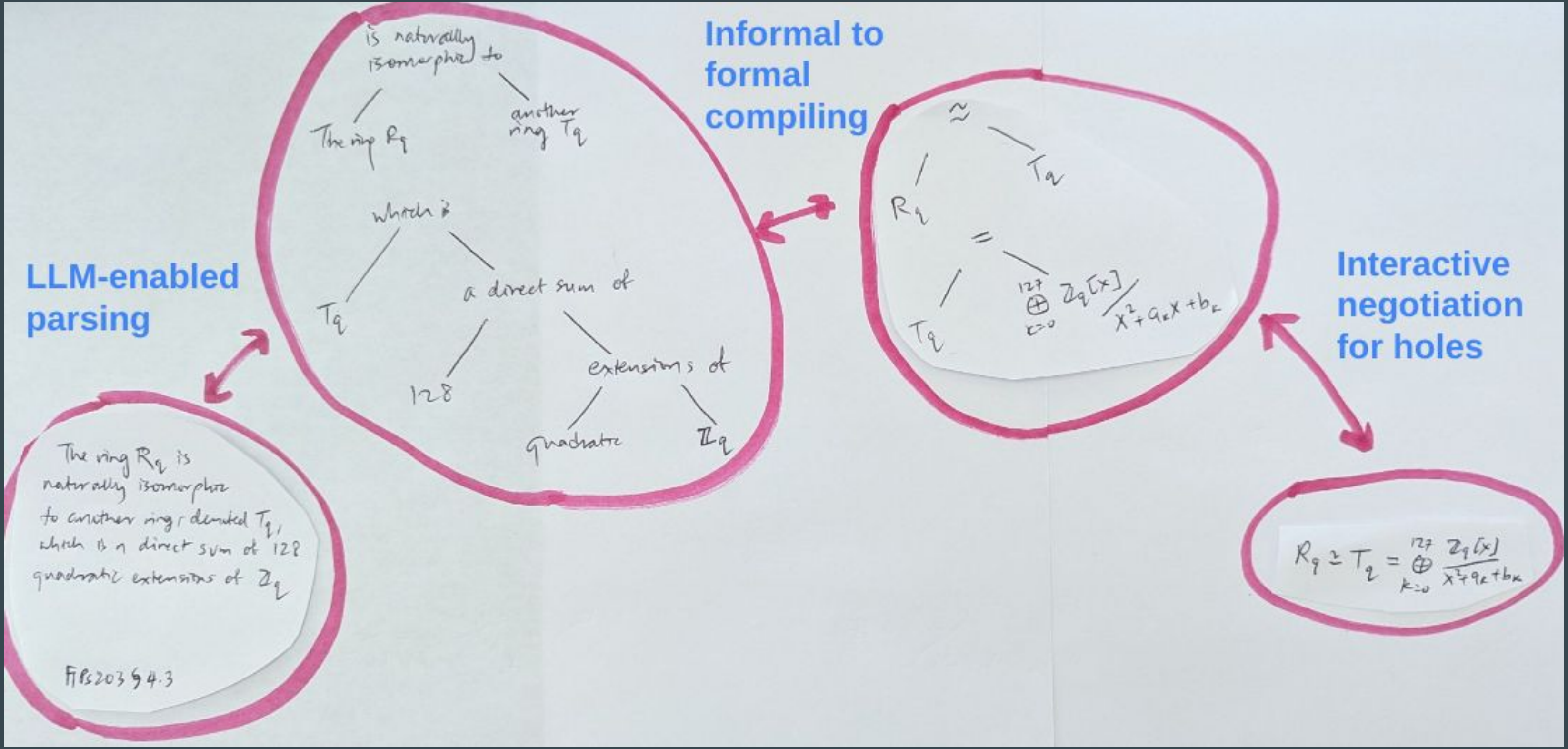
$$\text{if notKey}(k,l) \text{ then } \mathbf{ret}\ [(k,v)] \cup l \text{ else } \mathbf{ret}\ l$$

```
hone method insert. {
      StartMethod.
      setoid_rewrite refine_ReplaceUsedKeyAdd.
      setoid_rewrite refine_SubEnsembleInsert.
      autorewrite with monad laws.
      setoid_rewrite refine_pick_KeyToBeReplaced min.
      setoid_rewrite refine_If_Then_Else_Bind.
      autorewrite with monad laws.
      setoid_rewrite refine_If_Opt_Then_Else_Bind.
      autorewrite with monad laws.
      setoid_rewrite refine_pick_CacheADTwLogIndex AbsR.
      setoid_rewrite refine_pick_KVEnsembleInsertRemove
                              with (1 := EquivKeys H).
      setoid_rewrite refine_pick_KVEnsembleInsert
                              with (1 := EquivKeys H).
      autorewrite with monad laws; simpl.
      finish honing. }
```

Synthesis involves Refinement



Synthesis - not just for programs and proofs, but also for specs



Demos as Conversation Starters

http://18.118.1.7:8501/