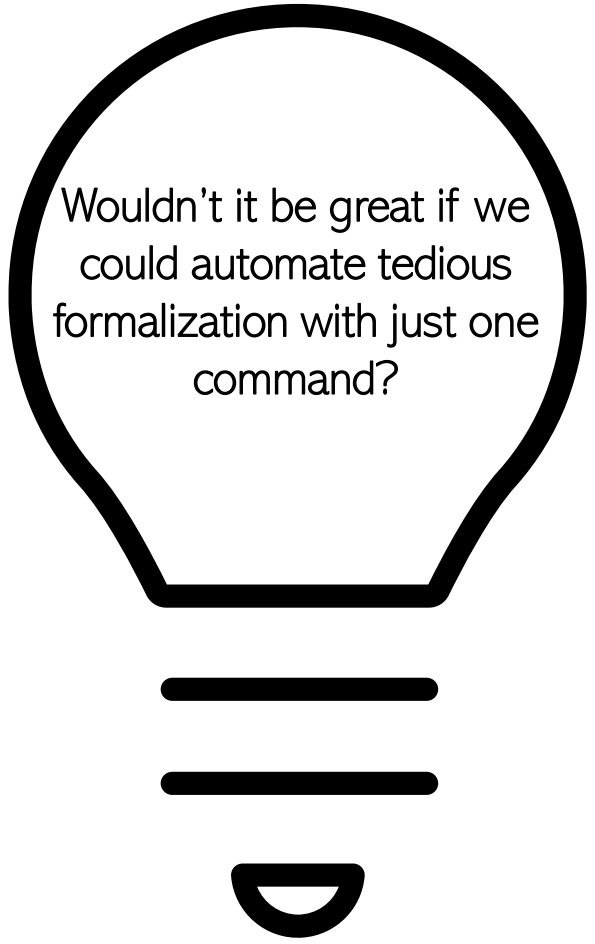
Lean-SMT: Integration of Theorem Provers with SMT Solvers

Haniel Barbosa, Clark Barrett, Tomaz Gomes, Harun Khan,
Abdalrhman Mohamed, Andrew Reynolds, Cesare Tinelli

Motivation

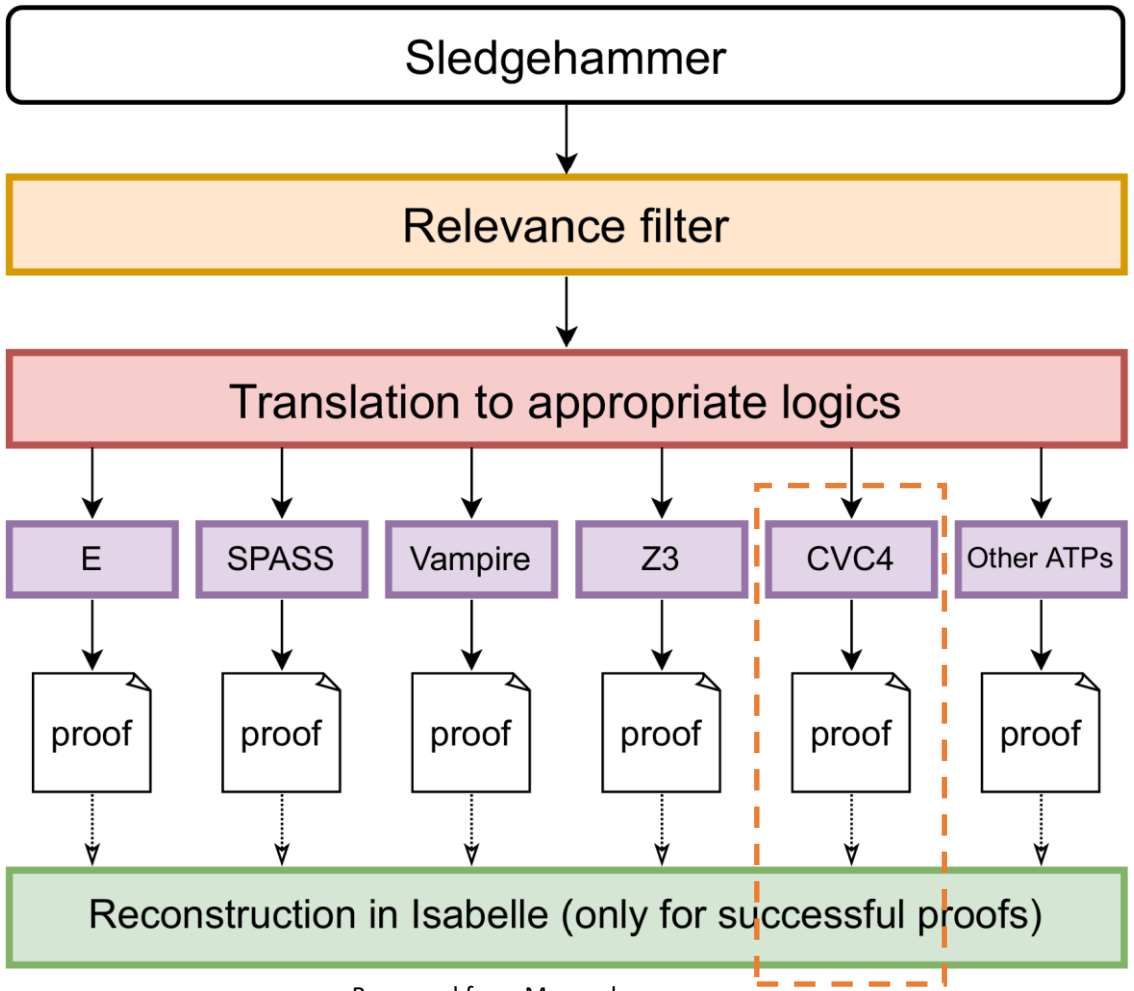
Suppose we encounter this in the Lean theorem prover:

```
x z: ℤ
hxz: x + z < 2
f: ℤ → ℤ
hz: 0 < z
hx: 0 ≤ x
⊢ ∀ (y : ℤ), f (x + y) = f y
```



Hammers

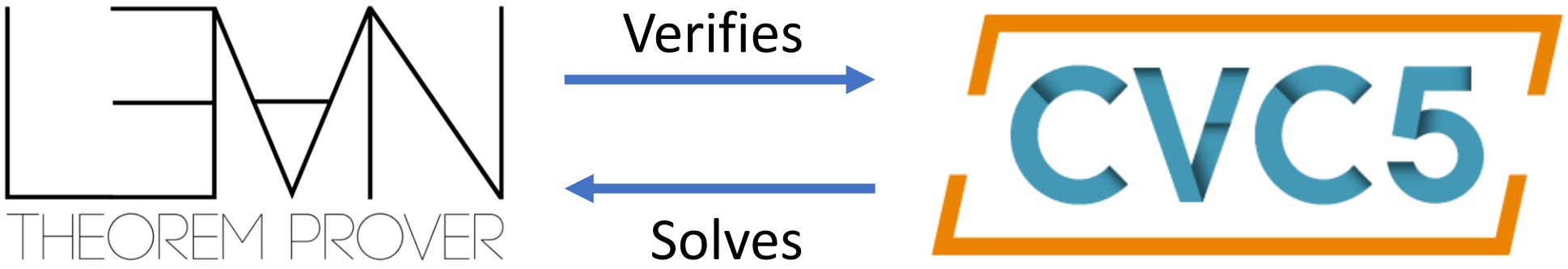
- Powerful automation tactics
  - 40% of Mizar Math library
  - 47% of Flyspeck libraries (HOL Light)
  - Similar results in Isabelle
- Integrate ATPs into proof assistants
  - SMT, Superposition, tableau
- Components
  - Premise selector
  - Translation module (into ATP's logic)
  - Proof reconstruction module
- Implement highlighted portion in Lean
- Use cvc5!



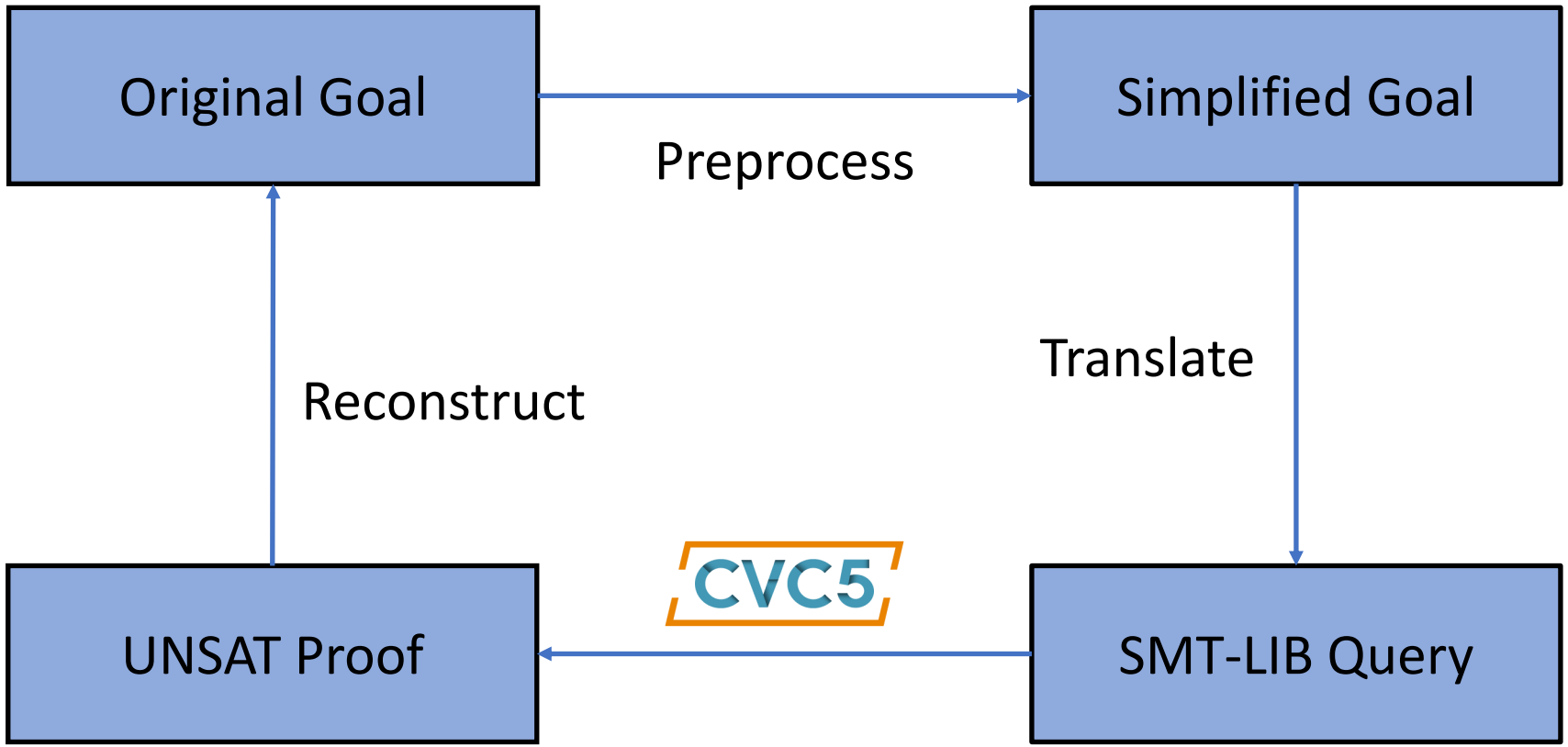
Borrowed from Magnushammer paper.

The Lean-SMT Tactic

A tactic that integrates cvc5 and Lean would benefit both worlds



Tactic Workflow



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
```

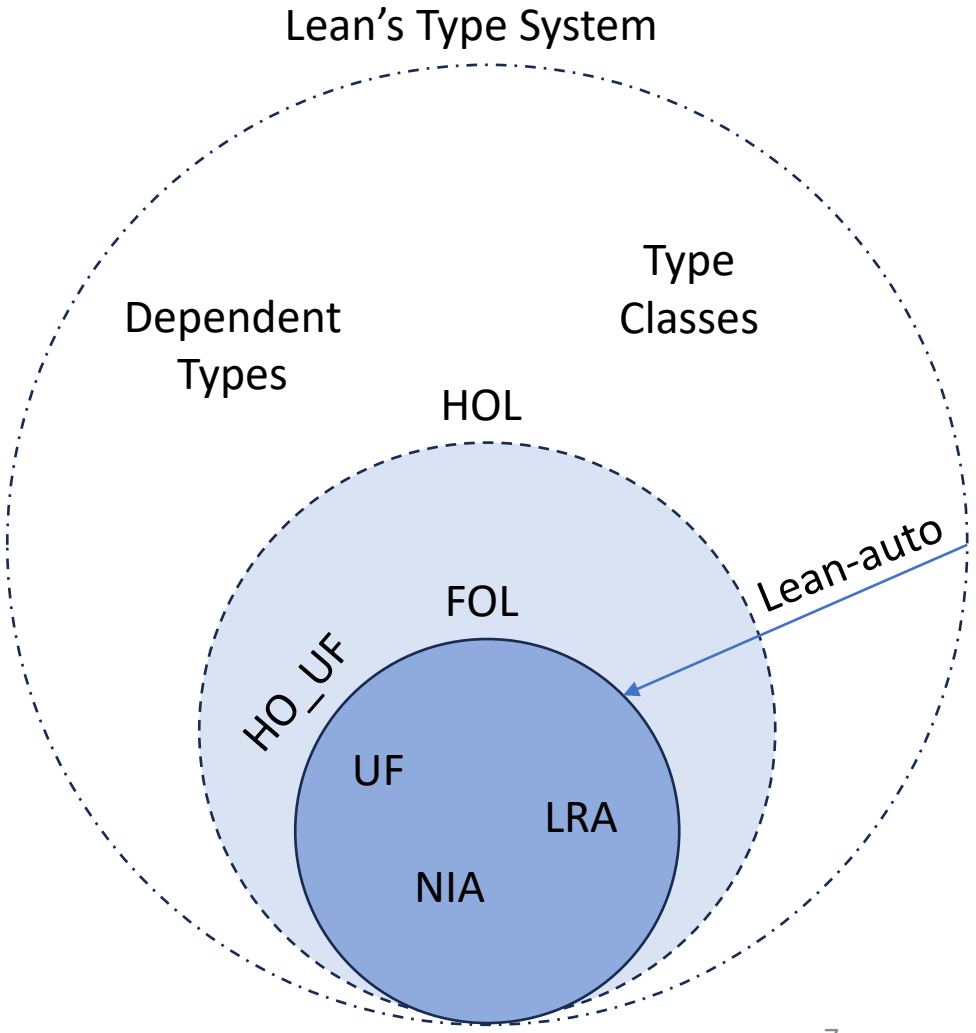
SMT-LIB does not support quantifying over Types and Type classes!!!

Identity element is unique in any group G!

Lean-SMT's Pipeline

Preprocessing → Translation → Solving → Reconstruction

- Lean's type system is very rich!
- SMT-LIB is based on FOL (with limited HOL)
- Utilize Lean-auto!
  - Very helpful, but not enough!
- 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.



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)
```

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)
```

Translation is sound because G is nonempty!

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)

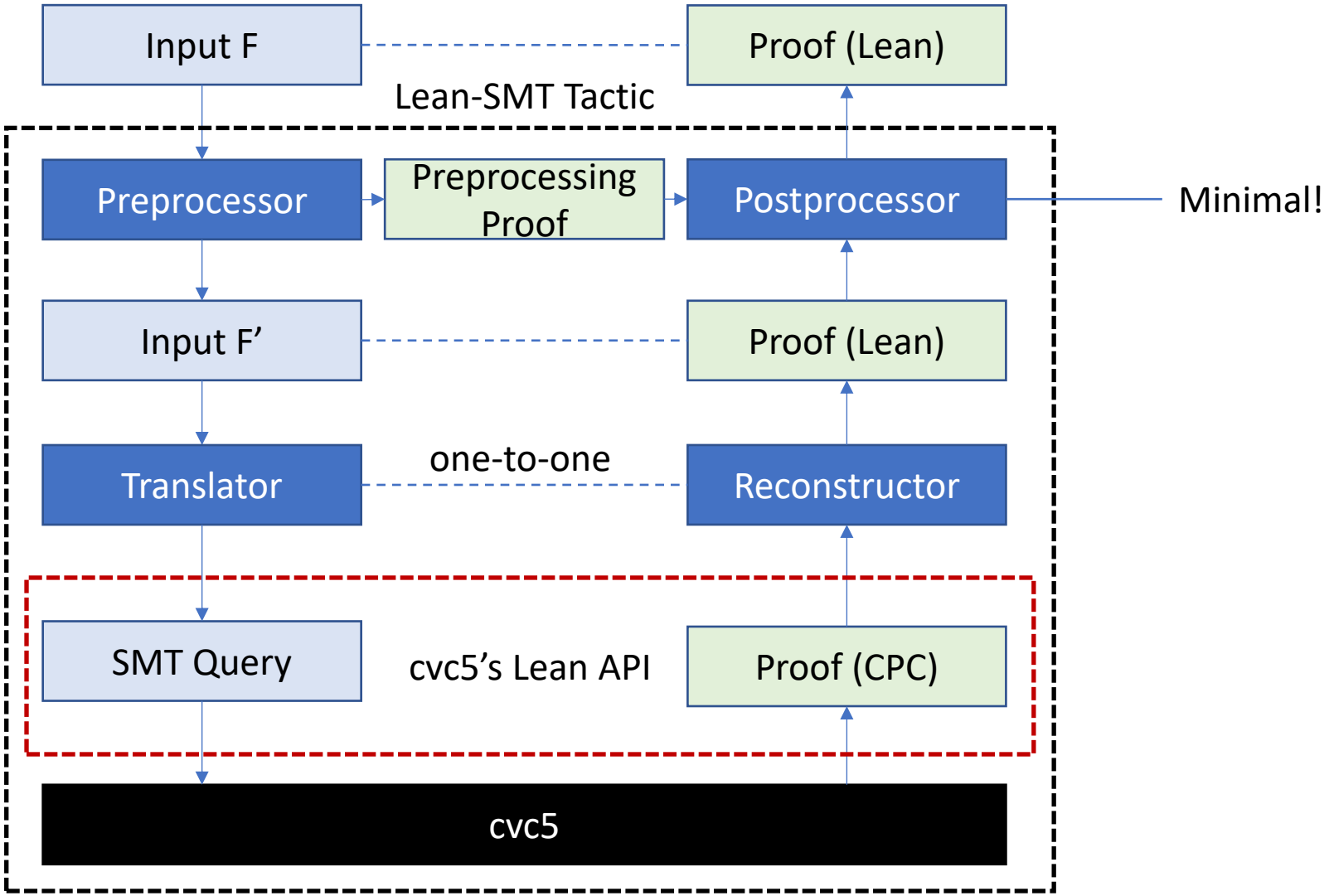
```
(define @t1 () (eo::var "a" G))
(define @t2 () (op e @t1))
...
(assume @p1 (forall @t3 (= @t2 @t1)))
(assume @p2 (forall @t3 (= @t4 e)))
(assume @p3 @t8)
(assume @p4 (not (= (forall @t3 (= @t9 @t1))
                    (= |e'| e))))
(step @p5 :rule quant-merge-prenex
          :args ((= @t8 @t10)))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



Reconstruction: Proof Replay vs. Verified Checker

Proof Replay

- Reconstructs proofs inference-by-inference inside the proof assistant
- Example: Isabelle SledgeHammer
- Pro: easy to implement and modify
- Con: slower
- Our approach!
  - No standard SMT proof format
  - cvc5 proof rules are not stable

Verified Checker

- Proofs are checked within the proof assistant yielding a proof by reflection rather than simulation
- Example: SMTCoq
- Pro: fast
- Con: hard to implement and modify

Proof Reconstruction

- A cvc5 proof is a sequence of steps (instantiations of proofs rules)
- There are ~130 proof rules and ~360 rewrite rules
- Proof and rewrite rules must be formalized in Lean!
- Three approaches:
  - Theorems: prove a theorem capturing a proof rule/rewrite rule
  - Tactics: repeated application of a set of theorems
  - Reflection: normalization steps

Proof Reconstruction: Theorems

The most straightforward approach is to formalize and prove a theorem capturing a proof rule/rewrite rule (e.g. **RESOLUTION**)

$$\frac{C_1, C_2 \mid pol, L}{C}$$

where

- C_1 and C_2 are CNF clauses
- pol is either $\top$ or $\bot$, representing the polarity of the pivot on the first clause
- L is the pivot of the resolution, which occurs as is (resp. under a NOT) in C_1 and negatively (as is) in if $pol = \top$ ($pol = \bot$)

```
theorem orN_resolution
  (hps : orN ps) (hqs : orN qs)
  (hi : i < ps.length)
  (hj : j < qs.length)
  (hij : ps[i] = ¬qs[j]) :
  orN (ps.eraseIdx i ++ qs.eraseIdx j)
```

Proof Reconstruction: Tactics

- In some cases where the formalization of the process is too complex, we write a tactic
- Example: **QUANT_MINISCOPE_AND** rewrite rule

$$\forall X. F_1 \wedge \cdots \wedge F_n = (\forall X. F_1) \wedge \cdots \wedge (\forall X. F_n)$$

- X denotes a multiple variables!
- This involves two key theorems

```
theorem miniscope_andN {ps : List (α → Prop)} :
  (∀ x, andN (ps.map (· x))) = andN (ps.map (∀ x, · x))
theorem forall_congr {α : Sort u} {p q : α → Prop} (h : ∀ a, p a = q a) :
  (∀ a, p a) = (∀ a, q a)
```

- Iteratively apply the two theorems, peeling one variable at a time

```
example : (∀ x y, p x y ∧ q x y) = ((∀ x y, p x y) ∧ (∀ x y, q x y)) := by
  smt
```

Proof Reconstruction: Reflection

- Verified programs allow us to prove correctness properties about an executable program
- `poly_norm` normalizes polynomials (up to distrib., assoc., and comm.)

```
example (x y : Int) (z : Real) : 1 * ↑(x + y) * z / 4 = 1 / (2 * 2) * (z * ↑y + ↑x * z) := by
  poly_norm
```

- `poly_norm` completely expands polynomials into a sum of monomials
- Variables in monomials are defined as `List Nat` allowing them to be compared easily
- Our implementation proves theorems pushing evaluation into each operator.

```
theorem denote_mul {m₁ m₂ : Monomial} : (m₁.mul m₂).denote ctx = m₁.denote ctx * m₂.denote ctx
theorem denote_add {p₁ p₂ : Polynomial} : (p₁.add p₂).denote ctx = p₁.denote ctx + p₂.denote ctx
```

- And a correctness theorem

```
theorem denote_eq_from_toPolynomial_eq {e₁ e₂ : RealExpr}
  (h : e₁.toPolynomial = e₂.toPolynomial) : e₁.denote ictx rctx = e₂.denote ictx rctx
```

Demo!

Current Status & Evaluation

- Supports UFs, linear integer/real arithmetic, and quantifiers
- No benchmarks for Lean (in progress!)
- Tested on Isabelle benchmarks from previous publications
- In SMT-LIB format (1 minute timeout)

Benchmark Set	Reconstruction Succeeded	Solved by cvc5	Total Number of Benchmarks
SledgeHammer	1722	1750	**4260**
17 Provers	2762	2812	**5000**
Total	**4484**	**4562**	**9260**

Next Steps

- Create a benchmark set for Lean!
- Support more theories (BitVec, Datatypes, etc.)
- Support more proof rules (current state: 180 rules)
- Add more preprocessing steps (reduce papercuts)
- Improve integration with Lean-auto
- Work towards a Sledgehammer in Lean
  - described as “the difference between walking and running” (L Paulson).

Questions?