Lean4Less: Translating Lean to Smaller Theories via an Extensional-to-Intensional Translation

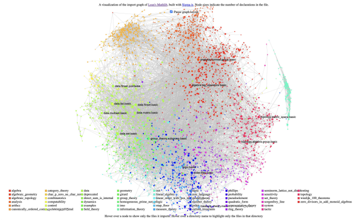
Rishikesh Vaishnav

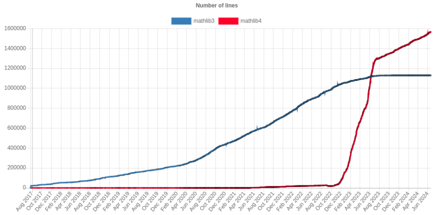
Presented at ICTAC 2025 in Marrakesh, Morocco

November 6, 2025

Introduction: Lean

- Lean (`https://lean-lang.org/`): proof assistant developed by the Lean FRO (`https://lean-fro.org/`)
- Type theory: calculus of inductive constructions with impredicative universe hierarchy
- mathlib4: large library of mathematics formalized in Lean 4


mathlib's import graph



mathlib's growth

Lean's Type Theory: Basics

Lean's type theory based on the Calculus of Inductive Construtions (CIC), with an infinite hierarchy of "type collections", i.e. `Sort`s, in particular:

- `Type` contains constructed mathematical concepts:
  - `(Nat : Type)`: type of natural numbers
  - `(Bool : Type)`: type of booleans
  - `(Real : Type)`: type of real numbers
  - `(List : Type → Type)`: type of lists of a specified type
- `Prop` contains logical statements:
  - `0 < 1`
  - `False → True`
  - `(l : List Nat).rev.rev = l`

Lean's Type Theory: Basics

Key to Lean's proof capabilities is the **propositions-as-types** principle:

```
-- any proposition implies itself
theorem ex1 (P : Prop) : P → P :=
fun (p : P) => p -- a function is a proof
```

- a logical implication statement in Lean is the same as a function type

Lean also features **dependent types**, enabling polymorphic functions:

```
-- defined for any `List T`
def List.append {T : Type} (xs ys : List T) : List T := ...
```

By propositions-as-types, dependent types in `Prop` are for-all statements:

```
-- all natural numbers are greater than or equal to 0
theorem ex2 (n : Nat) : n >= 0 := ...
```

Lean's Type Theory: Inductive Types

Lean's rich expressivity comes from the use of **inductive types**:

```
inductive Nat where -- the natural numbers
  | zero : Nat            -- zero, the smallest natural number
  | succ (n : Nat) : Nat -- the successor of a natural number `n`
```

- `Nat.succ` is a *recursive* constructor

Inductive types allow functions definitions via **pattern matching**:

```
-- the addition operation; `Nat.add a b` is abbreviated `a + b`
def Nat.add : Nat → Nat → Nat
  | a, Nat.zero   => a
  | a, Nat.succ b => Nat.succ (Nat.add a b)
```

These compile down to applications of **recursors** (a.k.a. eliminators)

```
-- `Nat.add` above elaborates to:
def Nat.add' (n m : Nat) : Nat :=
  Nat.rec n (fun _ ih => Nat.succ ih) m
```

- Recursive instances become inductive hypotheses (`ih` above)

Lean's Type Theory: Definitional Equalities

To aid in formalization, Lean also features certain **definitional equalities**:

```
inductive Vec : Nat → Type where
| nil : Vec Nat.zero
| cons : {n : Nat} → Vec n → Vec (Nat.succ n)
def ex3 (v : Vec n) : Vec (n + 0) := v
```

- The type of `v` is inferred as `Vec n`, but Lean identifies this type with `Vec (n + 0)`.

Definitional equality is utilized in Lean's "conversion" typing rule:

$$\frac{\Delta \vdash A, B : \texttt{Sort u} \quad \Delta \vdash A \equiv B \quad \Delta \vdash t : A}{\Delta \vdash t : B} \text{[CONV]}$$

- $\Delta \vdash t : T$ is Lean's typing judgment
- $\Delta \vdash a \equiv b$ is Lean's definitional equality judgment

Above, $\Delta \vdash$ `Vec n` $\equiv$ `Vec (n + 0)`, so $\Delta \vdash t :$ `Vec (n + 0)` by [CONV].

The Equality Type

It is possible to formlate an equality inductive type in Lean:

```
-- equality inductive type; `Eq a b` is abbreviated `a = b`
inductive Eq {A : Type} : A → A → Prop where
-- Eq.refl : {A : Type} → (a : A) → Eq a a
| refl (a : A) : Eq a a
```

By [CONV], we have `Eq.refl a : a = b` for any defeq a and b, e.g.:

```
theorem ex4 (n : Nat) : n + 0 = n := Eq.refl n
```

- Therefore, any definitional equality corresponds to a provable propositional equality

Where Definitional Equality Falls Short

Definitional equality may often be insufficient. Recall `Nat.add`:

```
def Nat.add : Nat → Nat → Nat
  | a, Nat.zero   => a
  | a, Nat.succ b => Nat.succ (Nat.add a b)
```

- This matches on the second argument, so while $n + 0$ and n are defeq, $0 + n$ and n are not (this isn't "obvious" to Lean)

  ```
  def ex5 (v : Vec n) : Vec (0 + n) :=
    v -- ERROR! expected type `Vec n`
  ```

Using ind. type elimination, we can prove equalities that aren't definitional:

```
theorem zero_add (n : Nat) : 0 + n = n :=
  match n with
  | .zero => rfl -- (`rfl` is short for `Eq.refl _`)
  | .succ n' =>
    let ih := zero_add n' -- inductive hypothesis: `0 + n' = n'`
    congrArg Nat.succ ih
```

- So, propositional equality is more powerful than definitional equality

Can we use this fact to manually "fix" `ex5` so Lean accepts it?

Explicit Type Conversion

We *can*, by using the `cast` operation (a.k.a. type transport):

```
def cast {A B : Sort u} (h : A = B) (a : A) : B := h.rec a
```

- `cast` allows you to type a term under a provably equal type
- Just as we have made equality explicit with `Eq`, we can make type conversion explicit with `cast`

To get Lean to accept our term, we wrap it with `cast` and a proof:

```
def ex5' (n : Nat) (v : Vec n) : Vec (0 + n) :=
  cast
    -- proof that `Vec n = Vec (0 + n)`
    (congr rfl (zero_add n).symm)
    v
```

- The cast around `v` is rather unsightly, but it works!

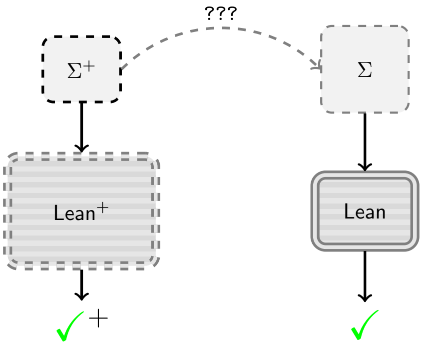
Speculation: A "Smarter" Kernel?

In effect, we have compensated for a *lack of expressivity* in Lean by using propositional equality

- Lean's defeq judgment is not quite as powerful as we would like
- Ideally, Lean would automatically "know" that `0 + n = n`, as soon as we have been able to prove it
- However, extending Lean's kernel to do this would make it more complex, possibly introducing bugs
- Lean's kernel is quite small and trusted, with some work being done in towards its formal verification; modifications are rare

Speculation: A “Smarter” Kernel?

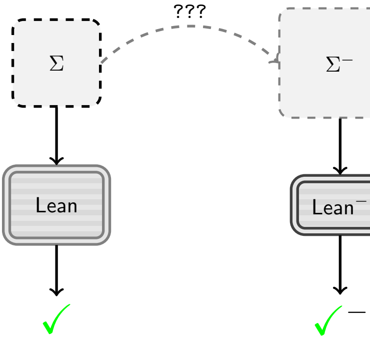
So, can we have the best of both worlds? Can we allow for *more definitional equalities* while maintaining proof verification via a small, reliable kernel?



- Perhaps, if we do this `cast`-based translation in a principled way

Speculation: Translation to a Weaker Theory?

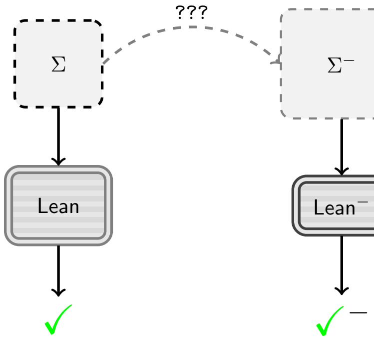
We can also think about the other direction: translating proofs to be verified with a *less powerful* kernel (fewer definitional equalities).



- Benefit: smaller kernels are easier to implement and verify, and are therefore more trustworthy

Speculation: Translation to a Weaker Theory?

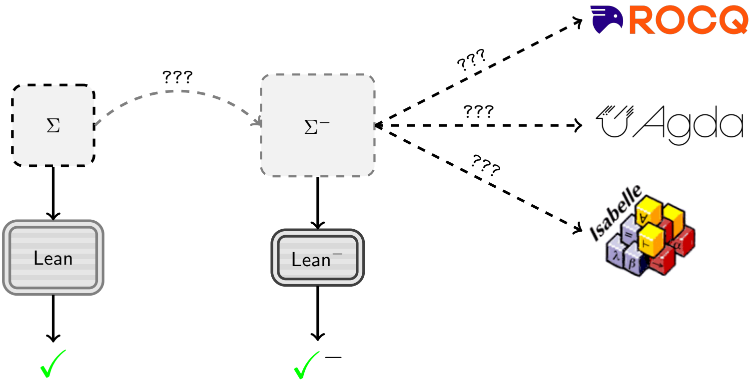
Another benefit of translation to a smaller theory: **proof translation**



- Easier proof export from smaller theory:
  - Fewer assumptions need to be made on target theory
  - Lower burden of encoding Lean-specific features in target system
- Verification with a smaller kernel prior to export

Proof Translation: Motivations

Why should we translate Lean to other systems?

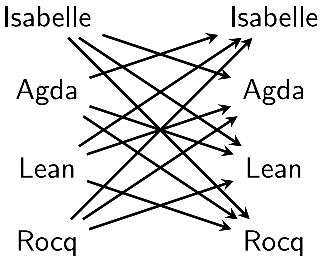


- Make Lean's formalizations available to them to extend/adapt
- Improve confidence in Lean's proof libraries through cross-checking
- Prevent duplication of work in writing libraries, tooling, etc.

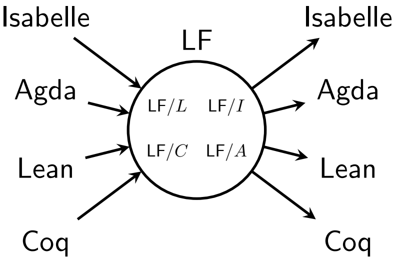
Dedukti (`https://deducteam.github.io/`): a logical framework specifically designed with translation in mind.

- Type system: lambda-pi calculus modulo rewrite rules ($\lambda\Pi/R$).
- Translation generally follows these steps:
  1. translate from theory A into Dedukti's encoding of A (DK/A)
  2. translate from DK/A to another compatible theory B (DK/B)
  3. translate from DK/B to B

Rather than $O(n^2)$ translations between proof assistants, go through a central logical framework:



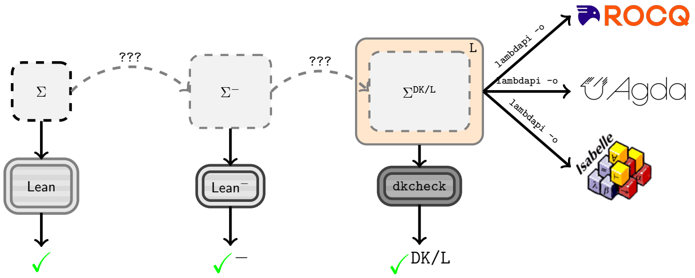
Naïvely



LF Approach

Proof Translation: Motivations

So, Dedukti can possibly fit into our translation as an intermediate theory:



- Dedukti encoding L accounts for defeqs specific to Lean^-

So: what defeqs should we eliminate in the initial translation?

- Whichever ones are *not directly encodable* within Dedukti
- It turns out that Lean's particular rules of "proof irrelevance" and "K-like reduction" do not give way to an encoding

Lean's Type Theory: Proof Irrelevance

Lean features a special defeq rule known as **proof irrelevance**:

$$\frac{\Delta \vdash P : \texttt{Prop} \qquad \Delta \vdash p, q : P}{\Delta \vdash p \equiv q}\ [\text{PI}]$$

- Any two proofs of the same proposition are identified

This is useful, for, instance, in subtyping:

```
-- type of `Nat`s less than 5
inductive LT5 : Type where
| mk : (n : Nat) → (p : n < 5) → LT5
theorem ex6 (n : Nat) (p1 p2 : n < 5) :
   LT5.mk n p1 = LT5.mk n p2 :=
  -- `p1` and `p2` are defeq by proof irrelevance, which gives us
  -- that `LT5.mk n p1` and `LT5.mk n p2` are defeq as well
  rfl
```

However, the typing requirement on p and q make this hard to encode.

- Dedukti's rewrite rules cannot "match" based on typing

Lean's Type Theory: K-Like Reduction

Lean features another special rule known as **K-like reduction**:

$$\frac{\Delta \vdash \texttt{mk}\ p_1\ \ldots\ p_n : \texttt{K}\ \ldots\ p_n\ \ldots \qquad \Delta \vdash t : \texttt{K}\ p_1\ \ldots\ p_n\ \ldots}{\Delta \vdash t \rightsquigarrow \texttt{mk}\ p_1\ \ldots\ p_n}\ \text{[KLR]}$$

This applies to any "K-like" type `K`, an inductive proposition with one constructor without arguments (except inductive type parameters)

- This is a **reduction rule**, not a definitional equality – it relates to the reduction subroutine of defeq-checking

As `Eq` is K-like, [KLR] allows the kernel to eliminate redundant casts:

```
theorem ex7 (n : Nat) (v : Vec n) (h : Vec n = Vec (n + 0)) :
    v = cast h v :=
  -- `v` and `cast h v` are defeq thanks to [KLR]
  rfl
```

[KLR] has a complex typing requirement on `t`, and is also hard to encode.

Our Target Theory: Lean$^-$

In our target theory Lean$^-$, we have removed [PI] and [KLR]:

$$\cancel{\dfrac{\Delta \vdash P : \texttt{Prop} \qquad \Delta \vdash p, q : P}{\Delta \vdash p \equiv q}}\ \text{[PI]} \qquad \cancel{\dfrac{\Delta \vdash \texttt{mk}\ p_1\ \ldots\ p_n : \texttt{K}\ \ldots \qquad \Delta \vdash t : \texttt{K}\ p_1\ \ldots}{\Delta \vdash t \rightsquigarrow \texttt{mk}\ p_1\ \ldots\ p_n}}\ \text{[KLR]}$$

Let's use $\Delta \vdash^- t : T$ for Lean's typing judgment.

- We will need to add proof irrelevance as an axiom:

```
axiom prfIrrel {P : Prop} (p q : P) : p = q
```

  as otherwise, there would be proofs we can no longer express.
- Our goal: define a translation $|\cdot|^-$ such that:

$$\text{If } \Delta \vdash t : T\text{, then } \texttt{prfIrrel} :: |\Delta|^- \vdash^- |t|^- : |T|^-.$$

  That is, typeability should be preserved by our translation (assuming `prfIrrel` in the Lean$^-$ context)

Defining a Translation: Existing Work?

So, how can we design and implement a translation eliminating these judgments? Is there any existing work that we can take advantage of?

- Relative to Lean$^-$, Lean is a theory where the axiom `prfIrrel` is "promoted" to a definitional equality
- Recall our difficulties around how `0 + n` is not defeq to `n`: what if we promoted *every* propositional equality to a definitional one?

This is the well-studied topic of extensional type theory (ETT), characterized by the "equality reflection rule":

$$\frac{\Delta \vdash_e A : \texttt{Sort u} \qquad \Delta \vdash_e t, s : A \qquad \Delta \vdash_e \_ : t \texttt{ = } s}{\Delta \vdash_e t \equiv s}$$

- Lean and Lean$^-$ are examples of "intensional" type theories (ITT)
- There is a good amount of existing work regarding translation from ETT to ITT – can we adapt this for our purposes?

Lean_e^-: an Extensional Theory

Suppose we add [RFL] to Lean^- to obtain an extensional theory Lean_e^-:

$$\frac{\Delta \vdash_e^- A : \texttt{Sort u} \qquad \Delta \vdash_e^- t, s : A \qquad \Delta \vdash_e^- \_ : t \texttt{ = } s}{\Delta \vdash_e^- t \equiv s}\ \text{[REFL]}$$

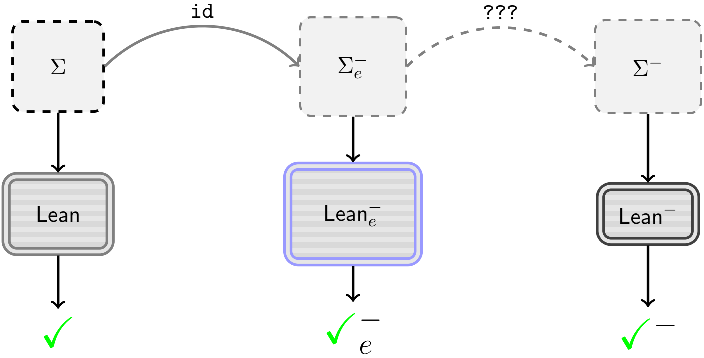
- Lean_e^- is strictly more expressive than Lean, since we can recover definitional proof irrelevance via [RFL]:

$$\frac{\Delta \vdash_e^- P : \texttt{Prop} \qquad \Delta \vdash_e^- p, q : P \qquad \Delta \vdash_e^- \texttt{prfIrrel p q} : p \texttt{ = } q}{\Delta \vdash_e^- p \equiv q}$$

(we can also show that [KLR] is recovered)

From Lean to Lean$^-$ via Lean$_e^-$?

Can we translate from Lean to Lean$^-$ using Lean$_e^-$ as a "middle ground"?



- Lean $\rightarrow$ Lean$_e^-$ is simply the identity function
- So, a Lean$_e^-$ $\rightarrow$ Lean$^-$ translation is also a Lean $\rightarrow$ Lean$^-$ translation!

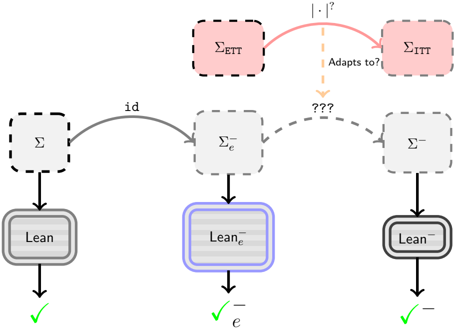
Theories Overview

To summarize, we have the following theories:

Theory	Rules	Axioms	$\subsetneq$
Lean$^-$ ($\vdash^-$)		prfIrrel	Lean
Lean ($\vdash$)	[PI], [KLR]		Lean$_e^-$
Lean$_e^-$ ($\vdash_e^-$)	[RFL]	prfIrrel	

Lean$_e^-$ $\to$ Lean$^-$ requires "eliminating" [RFL]: can this be done?

- Existing work on the translation from ETT to ITT may help



From ETT to ITT

What existing work on translating from ETT to ITT can we make use of?

- First conservativity result of ETT over ITT shown by Hofmann [3]
- A *constructive* proof first shown by Winterhalter et. al. [1]
    - Formalized in Rocq in the repository `ett-to-itt` [4]

The translation by Winterhalter et. al. takes *typing derivations* as input, and works with the more general **heterogeneous equality** type:

```
inductive HEq : {A : Sort u} → A → {B : Sort u} → B → Prop where
| refl (a : A) : HEq a a
```

- Allows for non-defeq LHS and RHS types

So, can we use `ett-to-itt` to translate Lean $\to$ Lean$^-$? Some problems:

- `ett-to-itt` is defined w.r.t. very specific ETT and ITT theories
- We have no way to access typing derivations in Lean!

Modifying a Typechecker

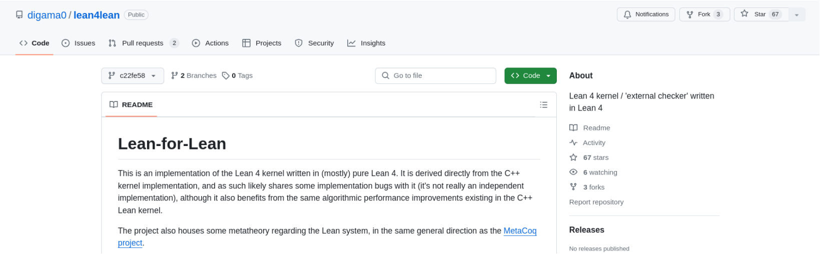
How can we get at the typing derivations needed for our translation?

Idea: repurpose a kernel typechecker.

- Typecheckers implement a “search” for valid typing derivations
- Steps can be correlated with uses of typing rules – our “input”!

Lean4Lean [2]: project to verify Lean’s kernel & formalize its meta-theory

- Contains a Lean typechecker kernel that is *implemented in Lean*:
  - Allows us to use Lean’s existing utilities to build our translation output
  - Leaves the door open for a formally verified translation



Lean4Lean's Typechecker

The main two typechecking functions found in Lean4Lean are:

```
def inferType (e : Expr) : RecM Expr := ...
def isDefEq (t s : Expr) : RecM Bool := ...
```

- `inferType`: main typechecking function, returning the a term's type
  - Thows exception if input is ill-typed
- `isDefEq`: checks whether two types are convertible
  - Returns `true` or `false`
  - Called as a subroutine of `inferType` for type conversion checking
- The `RecM` monad enforces a recusion depth limit
  - Termination is not guaranteed for all well-typed inputs

Lean4Less implementation: The main functions

Our final translation, "Lean4Less", has been adapted from Lean4Lean. We have repurposed these functions to have the following types:

```
def inferType (e : Expr) : RecM (Expr × Option Expr) := ...
def isDefEq (t s : Expr) : RecM (Bool × Option Expr) := ...
```

Both functions have a new second return value of type `Option Expr`:

- `inferType` produces a translation in parallel to typechecking
  - Explicit type conversions via `cast` are applied to subterms as necessary
- `isDefEq` produces the proof of equality between the LHS and RHS types needed by the `cast`s inserted by `inferType`
- We return an `Option` because of an important optimization: we only produce translated terms/proofs *when necessary*

Lean4Less implementation: The congruence lemmas

Lean4Less includes a file containing preliminary definitions, notably including a set of "congruence lemmas" with the types:

```
-- application congruence
theorem appHEq {A B : Type u}
  {U : A → Type v} {V : B → Type v}
  {f : (a : A) → U a} {g : (b : B) → V b}
  {a : A} {b : B}
  (hAB : A = B)
  (hUV : (a : A) → (b : B)
    → HEq a b → HEq (U a) (V b))
  (hfg : HEq f g) (hab : HEq a b)
  : HEq (f a) (g b) := ...
```

```
-- lambda congruence
theorem lamHEq {A B : Type u}
{U : A → Type v} {V : B → Type v}
  (f : (a : A) → U a) (g : (b : B) → V b)
  (hAB : A = B) (h : (a : A) → (b : B)
  → HEq a b → HEq (f a) (g b))
  : HEq (fun a => f a) (fun b => g b) := ...

-- forall congruence
theorem forAllHEq {A B : Type u}
  {U : A → Type v} {V : B → Type v}
  (hAB : A = B) (hUV : HEq U V)
  : ((a : A) → U a) = ((b : B) → V b) := ...
```

These lemmas are used by `isDefEq` to construct its equality proofs

- Returned proof is a kind of "trace" on the defeq derivation

Results: Some example translations

For instance, the following proof requires [PI] to be well-typed:

```
variable (P : Prop) (p : P) (q : P) (T : P → Prop)
-- `T p` is defeq to `T q` (due to proof irrelevance)
def ex8 (t : T p) : T q := t
```

- Our translation wraps a cast around `t`, translating it to:
  ```
  def ex8' (t : T p) : T q := cast (congrArg T (prfIrrel p q)) t
  ```
- Need to use `prfIrrel` axiom to prove this (not possible otherwise)

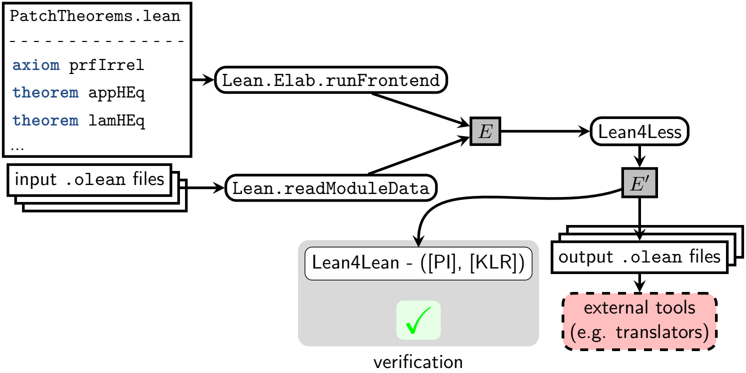
Translations can get quite complex, esp. involving dependent types:

```
variable (P : Prop) (Q : P → Prop) (p q : P) (Qp : Q p) (Qq : Q q)
variable (U : (p : P) → Q p → Prop)
def ex9 (t : U p Qp) : U q Qq := t
```

- This uses [PI] in a nested manner, leading to a larger translation:
  ```
  def ex9' (t : T p Qp) : T q Qq := cast (eq_of_heq
      (appHEq (congrArg Q (eq_of_heq (prfIrrel rfl p q)))
        (fun _ _ _ => HEq.rfl)
        (appHEq rfl ... HEq.rfl (prfIrrel rfl p q))
        (prfIrrel (congrArg Q (eq_of_heq (prfIrrel rfl p q)))
          Qp Qq))) t
  ```

Translation and verification workflow

Lean4Less generates and verifies its output as follows:



- Input environment E:
  - set of preliminary translation defs from file `PatchTheorems.lean`
  - constants from pre-elaborated source `.olean` files
- Output env. E': translated environment for export as `.olean` files
- Verification via a modified Lean$^-$ kernel (lacking [PI] and [KLR])

Results: Library translations

We have been able to translate (and verify) translations of the Lean standard library, as well as some smaller mathlib modules. Some numbers:

Module	Total Constants	Constants Using [PI]/[KLR] (% of total)	Input/Output Environment Size (Overhead)	Translation Runtime	Input/Output Typechecking Runtime (Overhead)[1]
`Std`	29859	1736/134 (6.3%)	226MB/261MB (15.5%)	18m02s	2m19s/3m9s (36.0%)
`Algebra.Order.Field.Rat`	113899	2965/237 (2.8%)	1485MB/1501MB (1.1%)	32m16s	5m11s/5m44s (10.6%)

- Translation output size overheads are reasonable
- Translation runtime is far from ideal, and comes coupled with high memory requirements that prevent us from effectively scaling
  - While the implementation is well-optimized in many respects, more investigation/work must be done to address this

Prospects: extensionality in Lean

Lean4Less's translation framework should be consistent with the general ETT to ITT translation (Winterhalter et al.)

- So, should be possible to extend to eliminate other definitional equalities (w/ new axioms/lemmas for each of them).
- This could include *new, user-defined* definitional equalities.
- While full ETT is undecidable, could add *partial* extensionality via a mechanism for registering/deriving new definitional equalities.

Could add a rule for "algorithmic reflection" to Lean:

$$\frac{\Delta \vdash^{-}_{e*} A : \texttt{Sort u} \qquad \Delta \vdash^{-}_{e*} t, u : A \qquad \Delta \vdash^{-}_{e*} \_ : t \;\texttt{=}\; u \text{ computable}}{\Delta \vdash^{-}_{e*} t \equiv u}$$

and extend Lean4Less to translate from this theory "Lean_{e*}".

Lean4Less could then be integrated with Lean's elaborator, allowing for reasoning modulo a extensible set of computable definitional equalities.

Prospects: Meta-theoretical simplifications

Lean$^-$ is a smaller theory, making it more feasible to prove **consistency**:

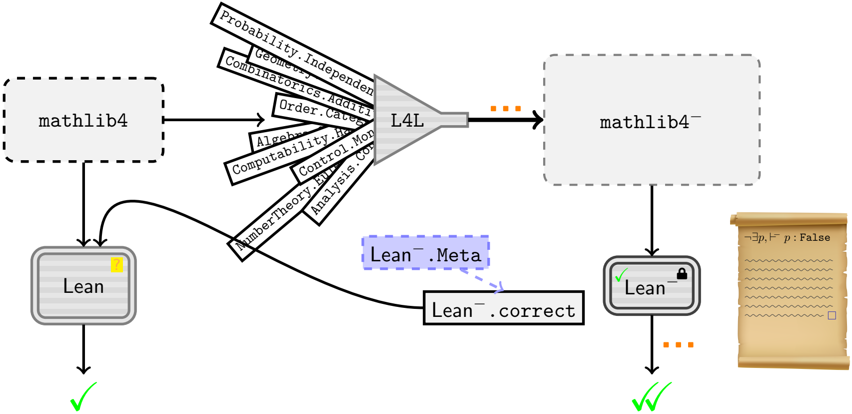
- Consistency property: there is no (axiom-free) proof of `False`
  - Important for ensuring that **proofs can be trusted**, i.e. kernel is "safe"
- Proof irrelevance and K-like reduction caused particular trouble in previous attempts w/ Lean's type theory; no longer exist in Lean$^-$

If Lean$^-$ is proven consistent, it becomes an ideal translation target:

- Can possibly use Lean4Less to translate proofs to be verified with a **provably safe kernel** deciding Lean$^-$
  - Requires a formal proof of correctness of this kernel implementation w.r.t. the Lean$^-$ theory
  - Such a kernel cannot possibly verify an axiom-free proof of `False`
- Can also extend translation to *eliminate more defeqs* that are problematic for the meta-theory, further restricting the Lean$^-$ theory

Prospects: Meta-theoretical simplifications

Ideally, we could translate entire libraries to Lean$^-$:



- Issue: translation does not currently scale very well
- Further optimizations may be possible, but seem tricky to implement

Prospects: Meta-theoretical simplifications

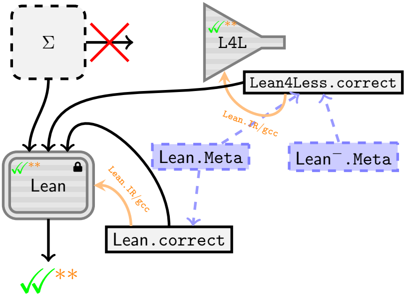
Alternatively, we could formally verify the following properties:

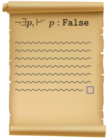
- The correctness of the translation implemented by Lean4Less
- The correctness of the Lean kernel w.r.t. the Lean theory

Thus, if Lean$^-$ is consistent (no proof of `False`), then so is Lean:

- Any proof of `False` in Lean translates to a proof of `False` in Lean$^-$

This justifies using a verified Lean kernel w/o translating entire libraries:



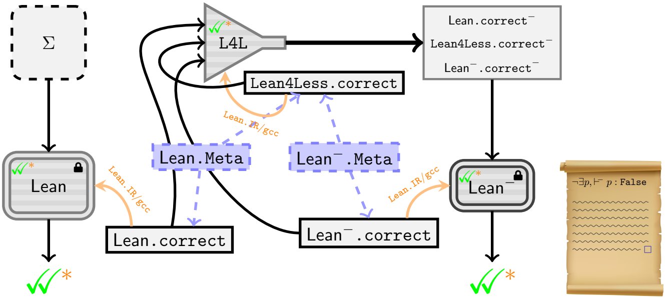


However, this carries some costs:

- Requires extra trust in code generators and compilers
- Lean kernel's consistency proof is partly checked by the *same* kernel

Prospects: Meta-theoretical simplifications

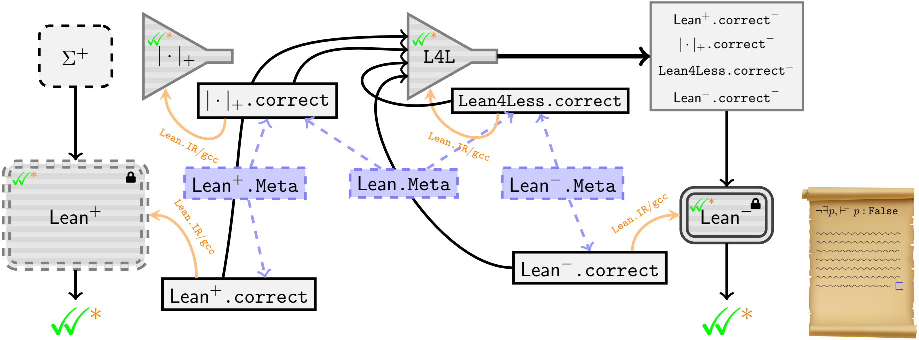
Ideally, we could also translate these correctness proofs to Lean$^-$:



- Can also show correctness of the Lean$^-$ kernel w.r.t. the Lean$^-$ theory
- Translation should be practical, but only enough to translate these select few proofs

Prospects: Meta-theoretical simplifications

Can also expand this approach to certain extensions of Lean:



- Can define a translation from some Lean$^+$ theory back to Lean
- If translation and Lean$^+$ kernel are verified, Lean$^+$ kernel is consistent

Conclusion

- Translating from Lean to smaller subtheories can be interpreted as a special case of a translation extensional to intensional type theory
- Such a translation *is possible* in practice, by modifying a kernel typechecker to construct translated terms
- Our translation, Lean4Less, implements the framework of a first (somewhat) practical translation from ETT to ITT that could possibly be extended to enable real-time extensional reasoning in Lean
- Verifying translation correctness could simplify meta-theoretical analyses of Lean relating to the consistency property, facilitating trust in future possible extensions of Lean's kernel

Thanks for listening!

References

[1] Théo Winterhalter et. al. "Eliminating reflection from type theory". In: *ACM SIGPLAN International Conference on Certified Programs and Proofs* (2019).

[2] Mario Carneiro. *Lean4Lean: Towards a formalized metatheory for the Lean theorem prover*. 2024. arXiv: 2403.14064 [cs.PL].

[3] Martin Hofmann. "Conservativity of Equality Reflection over Intensional Type Theory". In: *International Workshop on Types for Proofs and Programs*. 1995.

[4] Théo Winterhalter and Nicolas Tabareau. *ett-to-itt (Github)*.