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

Rish Vaishnav

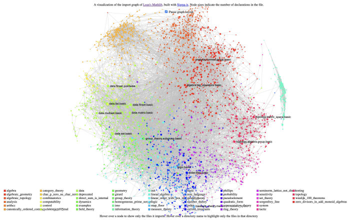
Presented at ICTAC 2025 in Marrakesh, Morocco

December 4, 2025

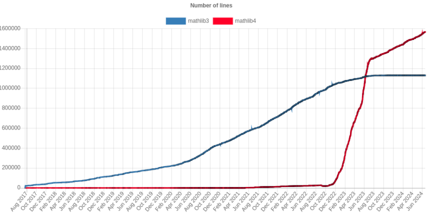
Introduction: Lean

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

THEOREM PROVER



mathlib's import graph



mathlib's growth

Lean's Type Theory: Basics

Lean's type theory is based on the Calculus of Inductive Construtions (CIC), w/ 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`

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
```

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`
```

Inductive types allow functions to be defined via **elimination**:

```
-- 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)
```

Lean's Type Theory: Definitional Equalities

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

```
inductive Vec : (n : Nat) → Type where ... -- vector of length `n`
def ex2 (v : Vec n) : Vec (n + 0) := v
```

- Type of `v` inferred as `Vec n`, which Lean identifies w/ `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, $\Delta \vdash \texttt{Vec n} \equiv \texttt{Vec (n + 0)}$, so $\Delta \vdash t : \texttt{Vec (n + 0)}$ by [CONV].

Propositional Equality

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 ex3 (n : Nat) : n + 0 = n := Eq.refl n
```

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

Explicit Type Conversion

The `cast` operation (a.k.a. type transport) makes conversion explicit:

```
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

The following definition is ill-typed in Lean:

```
def ex4 (n : Nat) (v : Vec n) : Vec (0 + n) :=
  v -- ERROR! expected type `Vec n`
```

- Problem: `0 + n` is not defeq to `n` (addition matches on second arg)

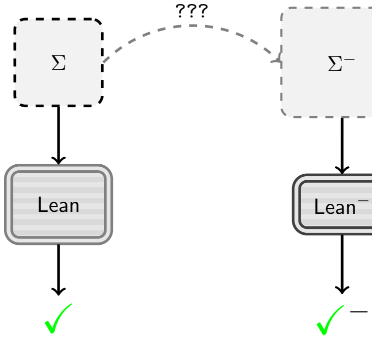
To get Lean to accept it, we wrap it with `cast` and a proof:

```
def ex4' (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: Translation to a Weaker Theory?

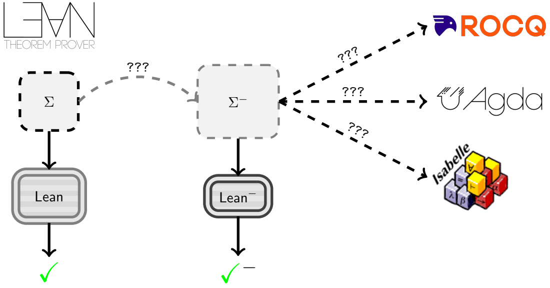
Above, we have compensated for a *lack of expressivity* in Lean by using `Eq` and `cast`. What if Lean was even *less* expressive (with fewer defeqs)? Can we generalize this process to translate to this smaller theory?



- 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: easier **proof translation**



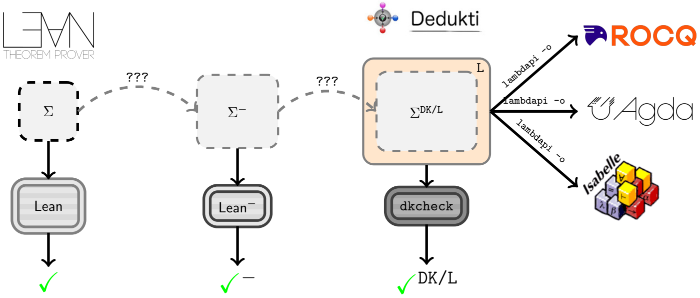
- Fewer assumptions need to be made on target theory

Some benefits of translation:

- 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.

Proof Translation: Motivations

We target Dedukti[1], a proof system and logical framework designed for interoperability, as an intermediate theory from which we export proofs



- 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

[1] https://deducteam.github.io/

Lean's Type Theory: Proof Irrelevance

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

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

- 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 ex5 (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**:

$$\frac{\Delta \vdash \texttt{mk}\ p_1\ \ldots\ p_n : \texttt{K}\ \ldots\ p_n\ i_1\ \ldots\ i_m \quad \Delta \vdash t : \texttt{K}\ p_1\ \ldots\ p_n\ i_1\ \ldots\ i_m}{\Delta \vdash t \rightsquigarrow \texttt{mk}\ p_1\ \ldots\ p_n}\ [\text{KLR}]$$

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

- 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 ex6 (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^{-} typing context)

Designing 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 [REFL] 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 [REFL]:

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

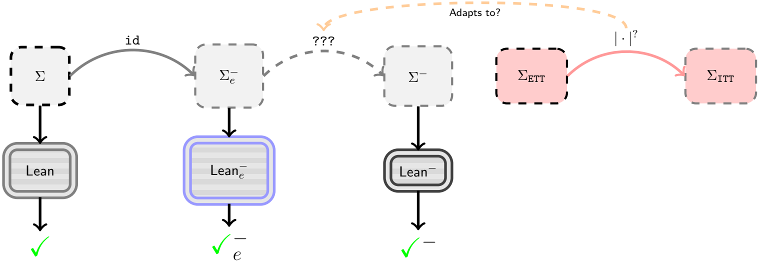
(we can also show that [KLR] is recovered)

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^- ($\vdash_e^-$)	[REFL]	`prfIrrel`	

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



- Lean $\to$ Lean_e^- is the identity function
- Lean_e^- $\to$ Lean^- requires "eliminating" [REFL]: 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!

The Lean4Lean kernel implementation

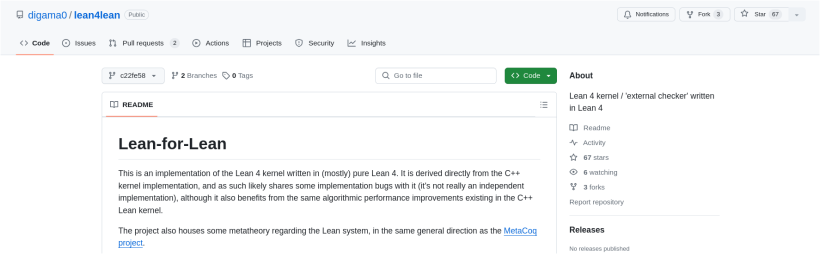
How can we get at the typing derivations needed for our translation?

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



Lean4Less implementation: The main functions

Our translation, "Lean4Less", has been adapted from Lean4Lean.
We repurpose Lean4Lean's main typechecking functions as follows:

```
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 type inference
  - Explicit type conversions via `cast` are applied to subterms as necessary
- `isDefEq` produces a proof of equality between the LHS and RHS when they are judged defeq
  - Needed by the `cast`s inserted by `inferType`
  - A sort of "trace" on the defeq derivation
- We return an `Option` because of an important optimization: we only produce translated terms/proofs *when necessary*

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 ex7 (t : T p) : T q := t
```

- Our translation wraps a cast around `t`, translating it to:
  ```
  def ex7' (t : T p) : T q := cast (congrArg T (prfIrrel p q)) t
  ```
- Need to use `prfIrrel` axiom to prove this (not possible otherwise)

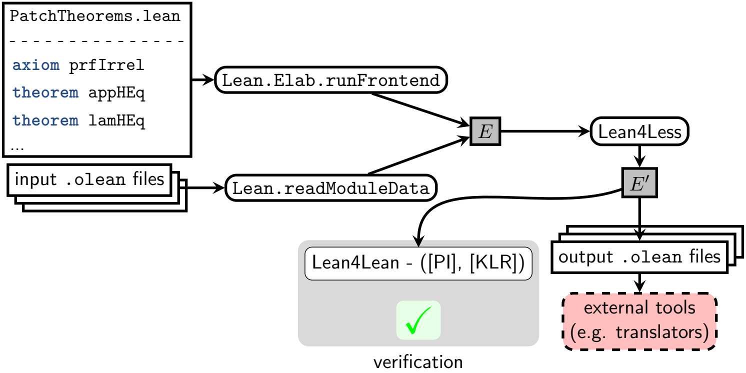
Translations can get quite complex, esp. involving dependent types:

```
variable (P : Prop) (Q : P → Prop) (p q : P) (Qp : Q p) (Qq : Q q)
variable (U : (p : P) → Q p → Prop)
def ex8 (t : U p Qp) : U q Qq := t
```

- This uses [PI] in a nested manner, leading to a larger translation:
  ```
  def ex8' (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)[2]
`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:

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

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.

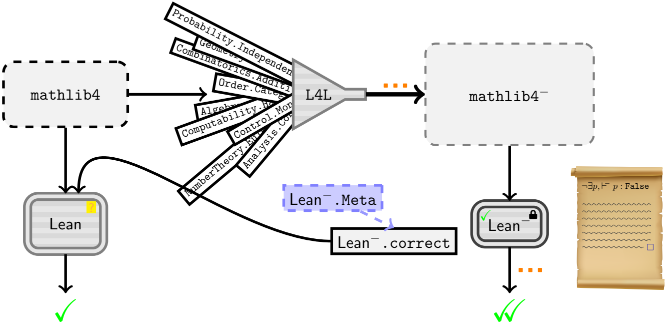
Prospects: Meta-theoretical simplifications

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"

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$^-$:



  - Issue: translation does not currently scale very well

Prospects: Meta-theoretical simplifications

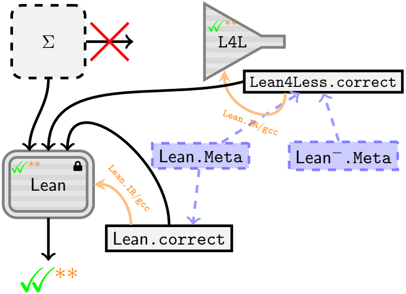
Alternatively, we could formally verify the following properties:

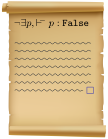
- The correctness of the translation implemented by Lean4Less
- The correctness of the Lean kernel w.r.t. the Lean theory

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:



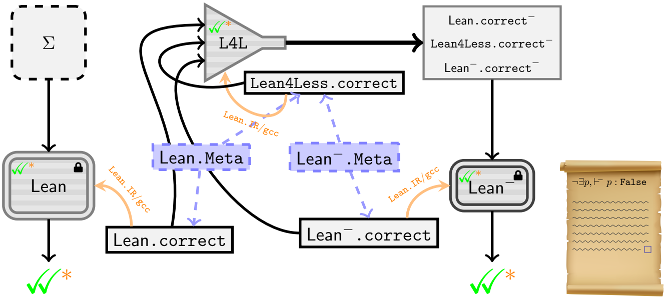


However, some caveats:

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

Prospects: Meta-theoretical simplifications

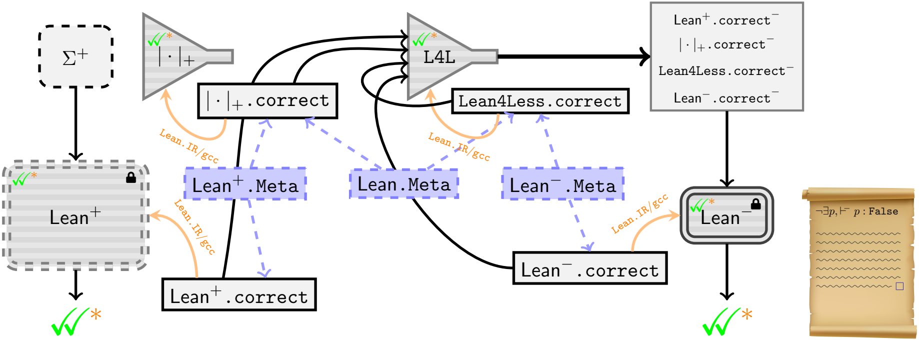
Ideally, we could also translate these correctness proofs to Lean$^-$:



- 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

To conclude:

- 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

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)*.

The End

Thanks for listening!