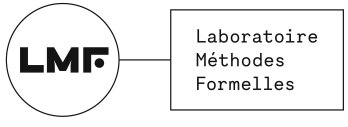
Lean4Less: Translating Lean to Smaller Theories via an Extensional to Intensional Translation

Rishikesh Vaishnav

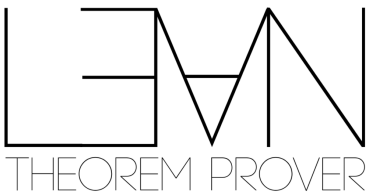
Presented at the LMF Nonpermanent Members Seminar

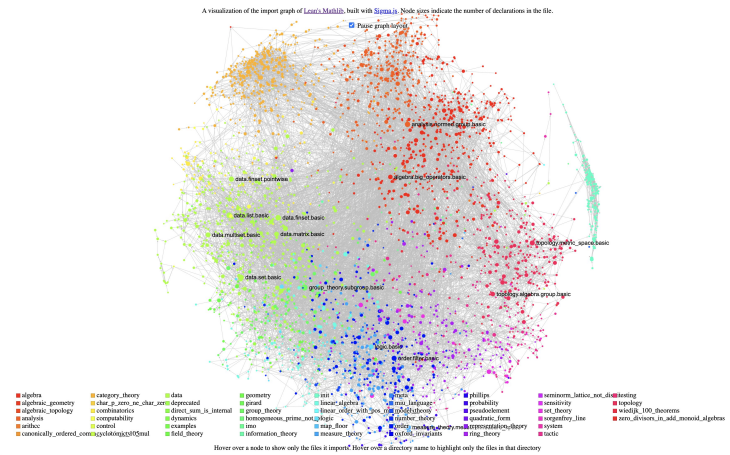
February 4, 2025



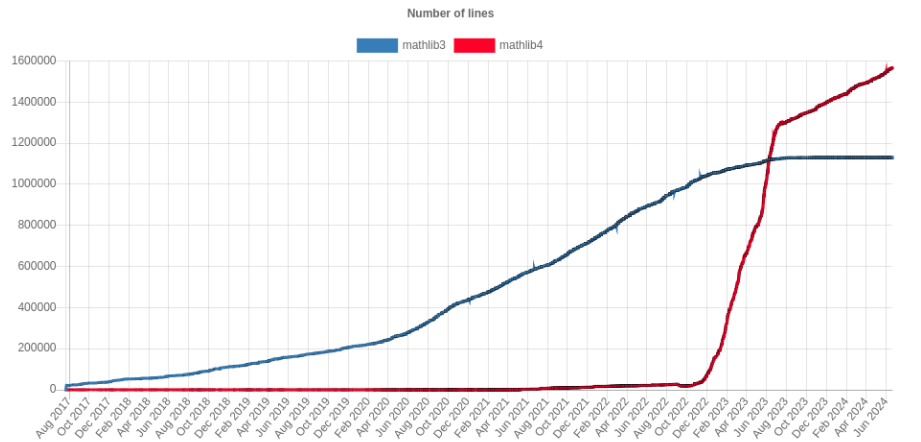
Introduction: Lean

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





mathlib's import graph



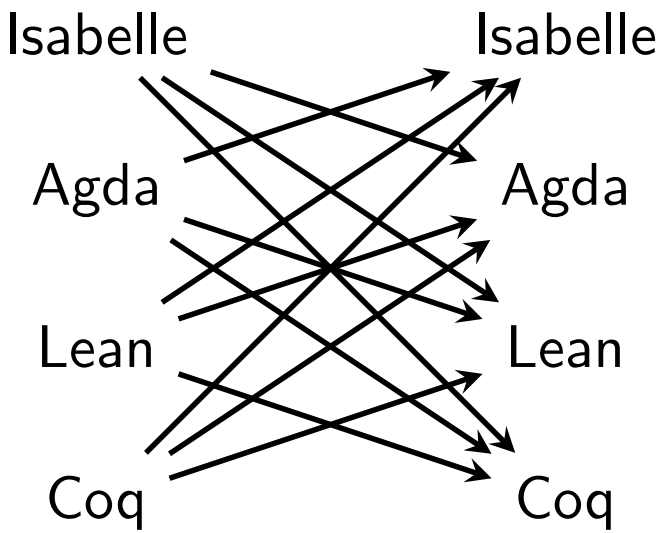
mathlib's growth

Introduction: Translation

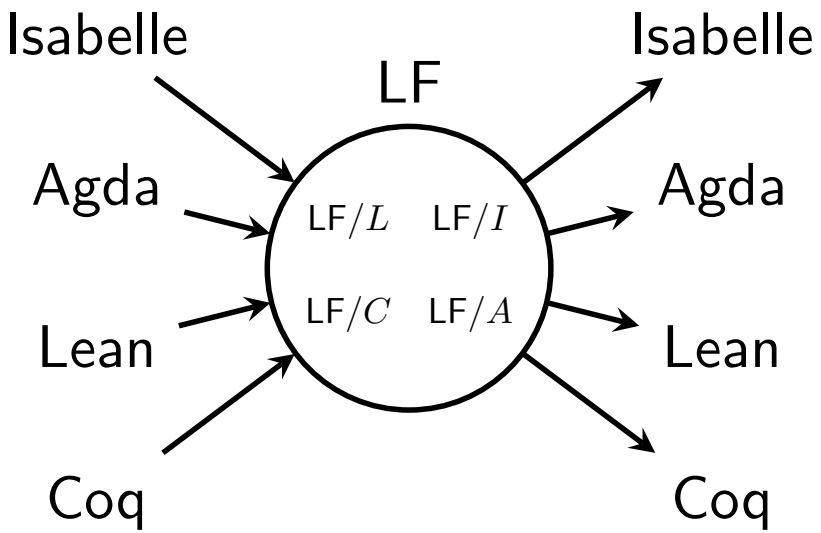
Some reasons why we would like to translate Lean to other systems:

- to make the large number of formalizations being done in Lean available to other systems (e.g. Coq, Agda, Isabelle)
- improve confidence in Lean's proof libraries by cross-checking them with other proof assistants
- prevent duplication of work in writing libraries, tooling, etc.

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



Naïvely



LF Approach

Introduction: Dedukti

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

- Type system: lambda-pi calculus modulo rewrite rules ($\lambda\Pi/R$). Definitional equality: normal forms via β-reduction + rewriting.
- 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

Lean's Type Theory (Algorithmic)

Lean's "algorithmic" judgments:

$$\frac{\Gamma \Vdash e : A}{\Gamma \Vdash e \Leftrightarrow e} \qquad \frac{\Gamma \Vdash e \Leftrightarrow e'}{\Gamma \Vdash e' \Leftrightarrow e} \qquad \frac{e \rightsquigarrow k \quad \Gamma \Vdash k \Leftrightarrow e'}{\Gamma \Vdash e \Leftrightarrow e'}$$

$$\frac{\ell \equiv \ell'}{\Gamma \Vdash \mathsf{U}_\ell \Leftrightarrow \mathsf{U}_{\ell'}} \qquad \frac{\Gamma \Vdash e_1 \Leftrightarrow e_1' : \forall x : A.\ B \quad \Gamma \Vdash e_2 \Leftrightarrow e_2' : A}{\Gamma \Vdash e_1\ e_2 \Leftrightarrow e_1'\ e_2'}$$

$$\frac{\Gamma \Vdash A : \mathsf{U}_\ell \quad \Gamma \Vdash e : B}{\Gamma, x : A \Vdash e : B} \qquad \frac{\Gamma \Vdash A : \mathsf{U}_\ell}{\Gamma, x : A \Vdash x : A} \qquad \frac{}{\Gamma \Vdash \mathsf{U}_\ell : \mathsf{U}_{S\ell}}$$

$$\frac{\Gamma \Vdash A \Leftrightarrow A' \quad \Gamma, x : A \Vdash e \Leftrightarrow e'}{\Gamma \Vdash \lambda x : A.\ e \Leftrightarrow \lambda x : A.\ e'} \qquad \frac{\Gamma \Vdash A \Leftrightarrow A' \quad \Gamma, x : A \Vdash B \Leftrightarrow B'}{\Gamma \Vdash \forall x : A.\ B \Leftrightarrow \forall x : A.\ B'}$$

$$\frac{\Gamma \Vdash e : \forall x : A.\ B \quad \Gamma \Vdash e' : A}{\Gamma \Vdash e\ e' : B[e'/x]} \qquad \frac{\Gamma, x : A \Vdash e : B}{\Gamma, x : A \Vdash \lambda x : A.\ e : \forall x : A.\ B}$$

$$\frac{\Gamma \Vdash e : \forall x : A.\ B \quad \Gamma, x : A \Vdash e\ x \Leftrightarrow e'\ x}{\Gamma \Vdash e' \Leftrightarrow e} \qquad \frac{\Gamma, x : A \Vdash e : B \quad \Gamma \Vdash e' : A}{(\lambda x : A.\ e)\ e' \rightsquigarrow e[e'/x]}$$

$$\frac{\Gamma \Vdash A : \mathsf{U}_{\ell_1} \quad \Gamma, x : A \Vdash B : \mathsf{U}_{\ell_2}}{\Gamma \Vdash \forall x : A.\ B : \mathsf{U}_{\mathrm{imax}(\ell_1, \ell_2)}} \qquad \frac{\Gamma \Vdash e : A \quad \Gamma \Vdash A \Leftrightarrow B}{\Gamma \Vdash e : B}$$

$$\frac{\Gamma \Vdash P : \mathsf{U}_0 \quad \Gamma \Vdash h : P \quad \Gamma \Vdash h' : P}{\Gamma \Vdash h \Leftrightarrow h'}\ (\mathsf{PI})$$

Lean's typing

Lean's definitional equality

The head reduction relation $a \rightsquigarrow b$ covers β- and recursor reduction.

In particular:

- conversion rule: typing up to definitional equality
- proof irrelevance rule: proofs are irrelevant for typing

Inductive Types and Recursors

Can also define "inductive types" that generate recursors and recursor reduction rules that are included in the defeq judgment:

```
inductive Nat : Type where
  | z : Nat
  | s : Nat → Nat

#check Nat.rec
-- Nat.rec.{u} {motive : Nat → Sort u}
--   (z : motive Nat.z) (s : (a : Nat)
--     → motive a → motive (Nat.s a))
--   (t : Nat) : motive t
```

The `Nat` inductive type

```
inductive Acc {A : Sort u} (r : A → A → Prop) : A → Prop
  where
  | intro (x : A) (h : (y : A) → r y x → Acc r y) : Acc r x

#check Acc.rec
-- Acc.rec.{u, v} {A : Sort v} {r : A → A → Prop}
--   {motive : (a : A) → Acc r a → Sort u}
--   (intro : (x : A) → (h : (y : A) → r y x → Acc r y)
--     → ((y : A) → (a : r y x) → motive y _) → motive x _)
--   {a' : A} (t : Acc r a') : motive a' t
```

The `Acc` inductive type

Checking Definitional Equality

Propositional equality in Lean is defined as follows:

```
inductive Eq : A → A → Prop where
  | refl (a : A) : Eq a a
#check (Eq.refl : {A : Sort u} → (a : A) → a = a)
```

where the type of `Eq.refl` expects the lhs and rhs to be defeq.

- We can use this to easily check whether two terms are defeq:

```
inductive Nat : Type where
| zero : Nat
| succ : Nat → Nat
```

```
def Nat.add (n m : Nat) : Nat :=
  -- (elaborates to a `Nat.rec` app)
  match n, m with
  | n, .zero   => n
  | n, .succ m' => Nat.succ (Nat.add n m')
```

```
-- provable with reflexivity (i.e., "by definition")
example (n : Nat) : n + 1 = Nat.succ n := Eq.refl (n + 1)

-- not provable with reflexivity (but still provable)
example (n : Nat) : 1 + n = Nat.succ n := _
```

Proof Irrelevance

The proof irrelevance rule allows us to ignore the content of proofs, only concerning ourselves with proof types when checking definitional equality.

- For instance, consider the Subtype inductive type:

```
inductive Subtype (A : Sort u) (p : A → Prop) where
| mk : (val : Nat) -> (property : p val) : Subtype A p
```

- Using this, we can define the following type:

```
def NatLT5 : Type := Subtype Nat (fun n => n < 5)

def NatLT5.mk (n : Nat) (p : n < 5) : NatLT5
  := @Subtype.mk Nat (fun n => n < 5) n p
```

- Proof irrelevance gives us the following definitional equality:

```
-- two different, non-defeq proofs that 3 < 5
theorem p1 : 3 < 5 := ...
theorem p2 : 3 < 5 := ...

-- (`rfl` is short for `Eq.refl _`)
theorem PIEx : NatLT5.mk 3 p1 = NatLT5.mk 3 p2 := rfl
```

K-Like Reduction

Consider the following inductive type:

```
inductive K : Prop where | mk : K
```

- Such "K-like inductive types" w/ a single constructor and no non-index arguments definitionally satisfy "axiom K" via PI:
  ```
  theorem K.axiomK (k : K) : k = K.mk := rfl
  ```
- By application congruence, we have the defeqs:
  ```
  #check (K.rec : {m : K → Sort u} → (mk : m K.mk) → (t : K) → m t)
  example (k : K) :
     @K.rec (fun _ => Bool) true k = @K.rec (fun _ => Bool) true K.mk
    := rfl
  example : @K.rec (fun _ => Bool) true K.mk = true := rfl
  ```
- However, we do not necessarily have:
  ```
  example (k : K) : @K.rec (fun _ => Bool) true k = true := rfl
  ```
  since Lean's typechecker does not (and cannot) implement transitivity.
- But in practice, we *do* have this defeq, thanks to "K-like reduction": the kernel is able to "rewrite" `k` to `K.mk` during reduction, allowing the LHS to reduce.

From Lean to Dedukti

Base translation: interpretation of Lean as a "Pure Type System"[1].
Additional rules must be translated to rewrite rules such that:

- they constitute a "confluent" system (i.e. every term has a unique irreducible/normal form)
- any two Lean-defeq terms have the same normal form

Rewriting is based on syntax matching – many of Lean's reduction/defeq rules are compatible, but not all.
Example of encoding Lean's `Nat` in Dedukti:

```
Lvl : Type.
z : Lvl.
s : Lvl -> Lvl.

Univ : Lvl -> Type.
El : s:Lvl -> Univ s -> Type.

Nat : Univ (s z).
zero : El (s z) Nat.
succ : El (s z) Nat -> El (s z) Nat.
```

```
def Nat_rec : (u : Lvl) ->
(motive : El (s z) Nat -> Univ u) ->
(zero : El u (motive zero)) ->
(succ : (n : El (s z) Nat) -> El u (motive n) ->
        El u (motive (succ n))) ->
(n : El (s z) Nat) ->
El u (motive n).

[u, motive, cz, csucc, n]
Nat_rec u motive cz csucc (succ n)
--> csucc n (Nat_rec u motive cz csucc n).
[u, motive, cz, csucc] Nat_rec u motive cz csucc zero --> cz.
```

[1] Denis Cousineau and Gilles Dowek. "Embedding Pure Type Systems in the Lambda-Pi-Calculus Modulo". In: *Typed Lambda Calculi and Applications*. 2007.

More complex definitional equalities: proof irrelevance

Recall proof irrelevance:

$$\frac{\Gamma \Vdash P : \mathsf{U}_0 \quad \Gamma \Vdash h : P \quad \Gamma \Vdash h' : P}{\Gamma \Vdash h \Leftrightarrow h'} \text{ (PI)}$$

This rule is tricky to encode within Dedukti:

- It is not a reduction rule, so we must devise a rewrite rule such that any two proofs of the same type have the same normal form.
- We may convert the typing condition into a syntactic one by outputting "annotated" proofs:

```
axiom P : Prop
axiom p : P
axiom q : P

axiom T : P → Type

def ex (t : T p) : T q := t
```

```
def erase : (Prp : Univ z) -> El z Prp -> El z Prp.
erased : (Prp : Univ z) -> El z Prp.
[Prp, p] erase Prp p --> erased Prp.

P : Univ z.
p : El z P.
q : El z P.

T : El z P -> Univ (s z).
def ex : El (s z) (T (Erase P p)) -> El (s z) (T (Erase P q)).
[t] ex t --> t.
```

- However, this approach runs into typing/pattern matching issues.

More complex definitional equalities: K-like reduction

Here, we have `k : K a b 0`, so by PI Lean can "rewrite" it to `@K.mk a b`, enabling recursor reduction:

```
inductive K (a b : Nat) : Nat → Prop where
  | mk : K a b 0
#check K.rec
-- K.rec.{u} {a b : Nat}
--   {motive : (c : Nat) → K a b c → Sort u}
--   (mk : motive 0 (K.mk a b)) {c : Nat}
--   (t : K a b c) : motive c t

-- defeq because of K-like reduction
-- (do not need constructor application to reduce)
theorem KEx (a b : Nat) (h : K a b 0)
  : @K.rec a b _ 10 0 h = 10 := rfl

-- not defeq because K-like reduction can't be applied;
-- the type of `h` does not match that of `K.mk a b`
theorem KEx' (a b : Nat) (h : K a b 1)
  : @K.rec a b _ 10 1 h = 10 := _
```

```
K : Nat -> Nat -> Nat -> Univ z.
mk : (a : Nat) -> (b : Nat) -> El z (K a b zero).
def K_rec : (u : Lvl) ->
  (a : El (s z) Nat) ->
  (b : El (s z) Nat) ->
  (motive : (c : El (s z) Nat) ->
    El (K a b c) -> Univ u) ->
  mk : (El u (motive zero (mk a b))) ->
  (c : El (s z) Nat) ->
  k : (El z (K a b c)) ->
  El u (motive c k).

[u, a, b, motive, cmk, k]
  K_rec u a b motive cmk zero k
  --> cmk.
```

This happens to be simple enough for a rewrite rule.

However, this conversion is not possible in general: `0` could instead be an arbitrarily complex expression involving `a` and `b` and quickly run into the limitations of rewriting pattern matching.

Idea: use axioms in place of definitional equalities

For ex below to be correctly typed, Lean must apply PI:

```
variable (P : Prop) (p q : P) (T : P → Type)
-- `T p` is defeq to `T q` (due to proof irrelevance)
def ex1 (t : T p) : T q := t
```

So, we cannot directly translate this to Dedukti.

- However, one tool we have under our belt is the cast operation:

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

  allowing explicit transport of terms to other, prop.-equal types.
- Adding a proof irrelevance axiom, and an arg congruence principle:

  ```
  -- proof irrelevance, represented as an axiom
  axiom prfIrrel {P : Prop} (p q : P) : p = q
  theorem congrArg (f : A → B) {x y : A} (h : x = y)
    : f x = f y := ...
  ```
- We can "patch" the body to get around the need for PI:

  ```
  def ex1' (t : T p) : T q := cast (congrArg T (prfIrrel p q)) t
  ```
- Question: can this be done in general?

Our target theory: Lean$^-$

Goal: translate Lean terms into theory "Lean$^-$", where PI has been replaced by an axiom PI-:

$$\frac{\Gamma \Vdash P : \mathsf{U}_0 \qquad \Gamma \Vdash p, q : P}{\Gamma \Vdash p \Leftrightarrow q} \text{ (PI)}$$

$$\frac{}{\Gamma \Vdash \texttt{prfIrrel} : \forall (P : \mathsf{U}_0),\ (p, q : P).\ p =_P q} \text{ (PI-)}$$

where $=_P$ is the equality type between proofs of proposition P. This is provable in Lean by reflection + proof irrelevance (so Lean$^-$ $\subsetneq$ Lean).

"Promoting" Propositional Equalities

Relative to our theory Lean$^-$, Lean is a theory where the propositional equality `prfIrrel` has been "promoted" to a definitional one.

- What if we turned *every* propositional equality into a definitional one?
- This would make using Lean a bit more natural, for instance:

```
inductive Vec : Nat → Type where
| nil : Vec 0
| cons {n : Nat} (v : Vec n) (x : Nat) : Vec (n + 1)

-- since we can prove n + 1 = 1 + n,
-- we don't have to worry about the order
def vecTest (n : Nat) (v : Vec n) : Vec (1 + n) :=
  v.cons 1
```

- However, this is ill-typed in Lean. To fix this, we apply casting:

```
def vecTest (n : Nat) (v : Vec n) : Vec (1 + n) :=
  cast (congr rfl (addOneComm n)) (v.cons 1)
```

- This is reminiscent of what we just did to patch proof irrelevance, so is our task a special case of a translation from a more general theory?

Extensional Type Theory and the Reflection Rule

The theory in question is that of extensional type theory, where every propositional equality becomes definitional via a "reflection" rule:

$$\frac{\Gamma \Vdash_{\mathsf{e}} A : \mathsf{U}_\ell \qquad \Gamma \Vdash_{\mathsf{e}} t, u : A \qquad \Gamma \Vdash_{\mathsf{e}} \_ : t =_A u}{\Gamma \Vdash_{\mathsf{e}} t \Leftrightarrow u} \ (\mathsf{RFL})$$

Adding this rule to Lean to obtain the theory "Lean$_{\mathsf{e}}^{-}$", we can recover PI from PI-:

$$\frac{\Gamma \Vdash_{\mathsf{e}} P : \mathsf{U}_0 \qquad \Gamma \Vdash_{\mathsf{e}} p, q : P \qquad \Gamma \Vdash_{\mathsf{e}} \mathtt{prfIrrel}\ P\ p\ q : p =_P q}{\Gamma \Vdash_{\mathsf{e}} p \Leftrightarrow q}$$

Therefore, any Lean derivation can be translated to one in Lean$_{\mathsf{e}}^{-}$ by replacing all uses of PI with the above (so, Lean $\subsetneq$ Lean$_{\mathsf{e}}^{-}$).

Theories overview and translation plan

To summarize our different theories:

Theory	Rules	$\subsetneq$
Lean^- ($\Vdash^=$)	PI-	Lean
Lean ($\Vdash$)	PI	$\mathsf{Lean}_\mathsf{e}^-$
$\mathsf{Lean}_\mathsf{e}^-$ ($\Vdash_\mathsf{e}^=$)	PI-, RFL	

As we can easily translate from Lean to $\mathsf{Lean}_\mathsf{e}^-$, it is sufficient to translate from $\mathsf{Lean}_\mathsf{e}^-$ to Lean^-. This is exactly the task of translating from extensional type theory (ETT) to intensional type theory (ITT) via the elimination of RFL.

- An algorithm for this was described by Winterhalter et al.[2] and was formalized in Coq in `ett-to-itt`[3].

[2] Théo Winterhalter, Matthieu Sozeau, and Nicolas Tabareau. "Eliminating reflection from type theory". In: *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs* (2019).

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

Complex translations

Translating from Lean^{-}_{e} to Lean^{-} may seem "overkill" for eliminating PI alone, however it is probably necessary, as proofs themselves are terms that can appear arbitrarily within other terms (in particular, within types).

- Consider the following "nested" use of proof irrelevance:

```
variable (P : Prop) (Q : P → Prop) (p q : P) (Qp : Q p) (Qq : Q q)
axiom T : (p : P) → Q p → Prop
def ex2 (t : T p Qp) : T q Qq := t
```

- This has a more complex translation:

```
inductive HEq : {A : Sort u} → A → {B : Sort u} → B → Prop where -- heterogeneous equality
| refl (a : A) : HEq a a
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) := ...
theorem eq_of_heq {A : Sort u} {a a' : A} (h : HEq a a') : a = a' := ...
-- proven using `prfIrrel`
theorem prfIrrelHEqPQ {P Q : Prop} (h : P = Q) (p : P) (q : Q) : HEq p q := ...

def ex2' (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 strategy

Using `ett-to-itt` directly presents some difficulties:

- Input derivations from a **minimal extensional theory**; must translate Lean derivations (using temporary axioms + RFL for some rules).
- Will need to modify a Lean typechecker to output these derivations.
- `ett-to-itt` outputs terms from a **minimal intensional theory**; will have to translate back to Lean (removing uses of temporary axioms).
- Consequently, the output will be very large and contain many unnecessary casts and redundant proof terms.

It may be easier to modify a typechecker to "patch" terms as necessary in parallel to typechecking (should also allow for a minimal translation).

Lean4Lean

One promising implementation for us to modify is Lean4Lean[4], a recent port of Lean's C++ typechecker code to Lean 4. The functions of primary interest to us are these, found in the file `Typechecker.lean`:

```
-- type inference
def inferType (e : Expr) : RecM Expr := ...

-- definitional equality check
def isDefEq (t s : Expr) : RecM Bool := ...

-- weak-head normalization
def whnf (e : Expr) : RecM Expr := ...
```

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

Lean4Less: a patching typechecker

We will modify these functions to return an additional `Option Expr`:

```
def inferType (e : Expr) : RecM (Expr × Option Expr) := ...
                                        -- ^ patched `e`
def isDefEq (t s : Expr) : RecM (Bool × Option Expr) := ...
                                        -- ^ proof of `HEq t s`
def whnf (e : Expr) : RecM (Expr × Option Expr) := ...
                                    -- ^ proof of `HEq e (whnf e)`
```

Terms will be patched by `inferType` to have type casts (i.e. transports) "injected" as necessary using proofs constructed by `isDefEq` / `whnf`:

- when checking that constant values have their expected types
- in the app case (in `f a` where `f : A → B`, we need `A` defeq `inferType a`)
- in the let case (in `let x : T := v`, we need `T` defeq `inferType v`)
- places where certain expression head constructors are expected after calling `whnf` (e.g. `Expr.sort l` for lambda/forall domain types)

Lean4Less: a patching typechecker

Note: the modified `isDefEq` and `whnf` return *heterogeneous* equality proofs (`HEq` in Lean) – necessary because the LHS/RHS types may only be propositionally equal in Lean$^-$. The following "patching lemmas" (a.k.a. "congruence lemmas") will be crucial to us [5]:

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

-- proof irrelevance
axiom prfIrrel (P Q : Prop) (h : P = Q)
  (p : Q) (q : P) : HEq p q

-- 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) := ...
```

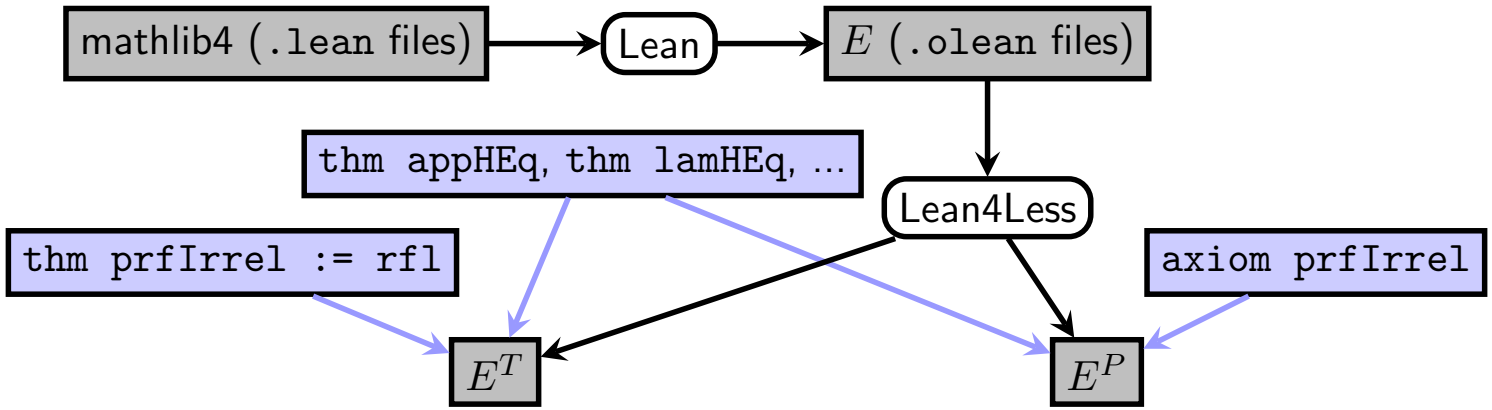
```
-- lambda congruence
theorem lamHEq {A B : Type u}
{U : A → Type v} {V : B → Type v}
  (f : (a : A) → U a) (g : (b : B) → V b)
  (hAB : A = B) (h : (a : A) → (b : B)
  → HEq a b → HEq (f a) (g b))
  : HEq (fun a => f a) (fun b => g b) := ...

-- 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) := ...
```

[5] see the full list of patching lemmas at `https://github.com/rish987/lean4lean/blob/ef65caba6ce4b5ee00d0955de4cda6807bd8c371/patch/PatchTheoremsAx.lean`

Testing plan

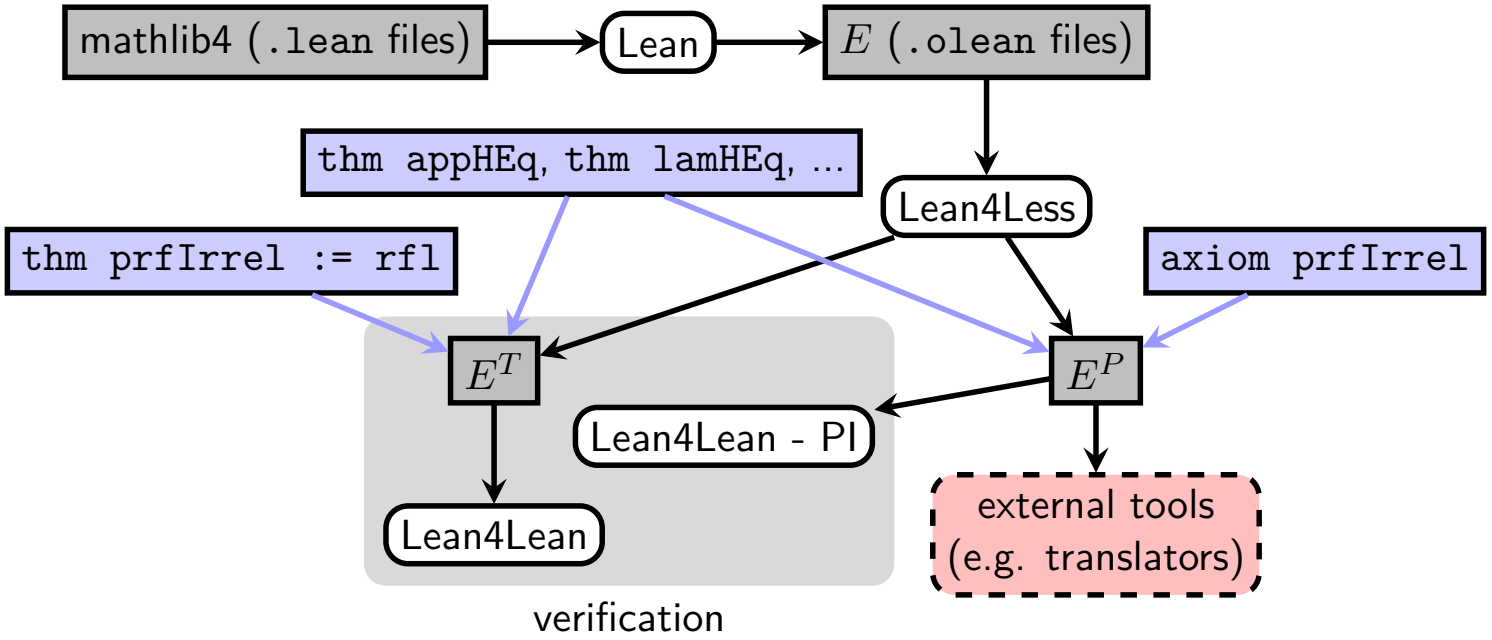
Once Lean4Less is implemented, we will test it on mathlib4:



- Input: mathlib `.olean` files from Lean (environment E)
- Outputs two sets of `.olean` files:
  1. E^P: the patched environment
  2. E^T: E + equality proofs between the original and patched types (for verification only)

Testing plan

The output environments will then be passed as input to other tools:



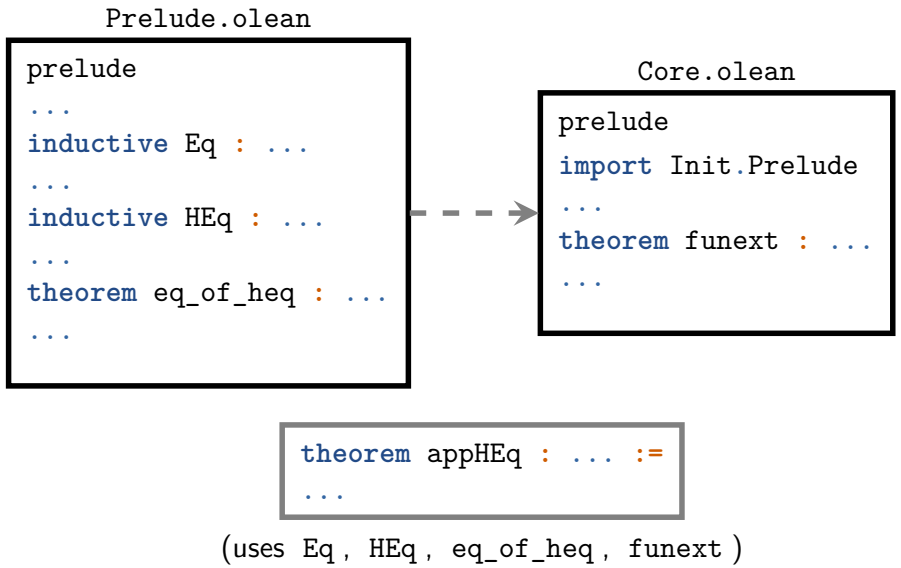
Verification steps:

- typecheck E^P w/ modified kernel representing Lean$^-$
- typecheck E^T w/ original kernel (checks that the "meaning" of types was preserved).

Patching lemma dependency extraction

A translated library can be output as a single `.olean` file, but this can be quite large and inconvenient to work with.

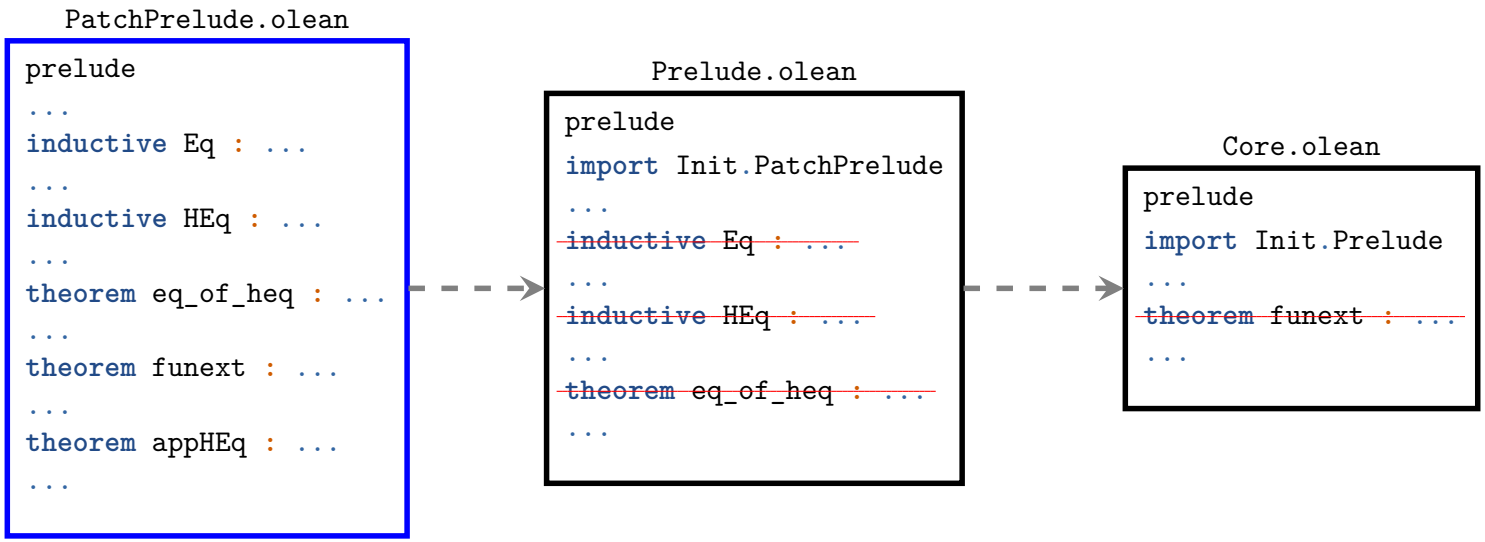
- Ideally, we would like to output separate `.olean` files using the same file structure from the input.
- However, patching lemmas depend on definitions strewn throughout the standard library:



- This makes it difficult to "place" the patching lemmas within existing `.olean` environments, as definitions between dependencies can (and in fact do) require patching.

Patching lemma dependency extraction

Solution: extract patching lemmas + dependencies to separate env output (as its own `.olean` file), and have the prelude env import it:



Optimization: Minimal patching lemma variants

To reduce the size of the output it may help to use variants of patching lemmas with fewer hypotheses where possible.

- E.g., uses of the fully general lemma

```
theorem appHEqABUV' {A B : Sort u}
 {U : A → Sort v} {V : B → Sort v} (hAB : HEq A B)
 (hUV : (a : A) → (b : B) → HEq a b → HEq (U a) (V b))
 {f : (a : A) → U a} {g : (b : B) → V b} {a : A} {b : B}
 (hfg : HEq f g) (hab : HEq a b)
 : HEq (f a) (g b) := ...
```

  can be simplified to uses of

```
theorem appHEqAB {A B : Sort u} {U : Sort v}
 (hAB : HEq A B)
 {f : (a : A) → U} {g : (b : B) → U} {a : A} {b : B}
 (hfg : HEq f g) (hab : HEq a b)
 : HEq (f a) (g b) := ...
```

  when output types are non-dependent and defeq in Lean$^-$

- Helps avoid redundant reflexivity proofs in the output

Optimization: Lambda-casting

Sometimes, we must apply a cast to a lambda expression:

```
axiom P : Prop
axiom Q : P → Prop
axiom p q : P
axiom X : (p : P) → Q p → Q p
theorem lamDemo : Q q → Q q := fun (qp : Q p) => X p qp
```

resulting in the translation:

```
theorem lamDemo : Q q → Q q :=
@L4L.castHEq (Q p → Q p) (Q q → Q q)
  (L4L.forallHEqAB (L4L.appArgHEq Q (L4L.prfIrrel P p q))
    (L4L.appArgHEq Q (L4L.prfIrrel p q)))
  fun (qp : Q p) => X p qp
```

We can "push" the cast into the lambda, obtaining more compact output:

```
theorem lamDemo : Q q → Q q
fun (qp : Q q) =>
  L4L.castHEq (L4L.appArgHEq Q (L4L.prfIrrel p q))
    (X p (L4L.castHEq (L4L.appArgHEq Q (L4L.prfIrrel q p)) qp))
```

Optimization: Application argument abstraction

Patching applications "as they are" can result in large outputs. E.g.:

```
axiom A : P → Nat → Nat → Nat → Nat → Nat → Nat → Prop
axiom Aq : A q 0 0 0 0 0 0

theorem absDemoA : A p 0 0 0 0 0 0 := Aq
```

naively translates to:

```
theorem absDemoA : A p 0 0 0 0 0 0 :=
L4L.castHEq (A q 0 0 0 0 0 0) (A p 0 0 0 0 0 0)
  (L4L.appFunHEq (A q 0 0 0 0 0) (A p 0 0 0 0 0) 0
    (L4L.appFunHEq (A q 0 0 0 0) (A p 0 0 0 0) 0
      (L4L.appFunHEq (A q 0 0 0) (A p 0 0 0) 0
        (L4L.appFunHEq (A q 0 0) (A p 0 0) 0
          (L4L.appFunHEq (A q 0) (A p 0) 0
            (L4L.appFunHEq (A q) (A p) 0
              (L4L.appArgHEq A q p (L4L.prfIrrel q p)))))))
  Aq
```

Optimization: Application argument abstraction

However, note that the application `A p 0 0 0 0 0 0` is equivalent to the application `(fun (x : P) => A x 0 0 0 0 0 0) p`.

- By optimizing the translation to perform this application abstraction when possible (prior to constructing the equality proof), we can obtain a much more compact output:

```
theorem absDemoA : A p 0 0 0 0 0 0 :=
L4L.castHEq (L4L.appArgHEq (fun (a : P) => A a 0 0 0 0 0 0)
  (L4L.prfIrrel P q p)) Aq
```

- In particular, the number of lemmas that need to be applied no longer depends on the number of Lean^{-}-defeq application arguments.
- Further considerations arise when we account for functions with dependent types/arity.
- This can likely be generalized to other expression types (i.e., abstracting subterms of lambda and forall expressions), though this has not been implemented yet (it would require some refactoring)

Scaling difficulties: Transport Hell

There are cases when we enter "transport hell", e.g. when expected types require translation themselves.

- This occurs in the following example:

```
inductive K : Prop where | mk : K
def F : Bool → Type | true => Bool | _ => Unit
structure S : Type where
  b : Bool
  f : F b
def projTest {B : Bool → Type} (s : B true)
   : B (@K.rec (fun _ => S) (S.mk true true) k).2 := s
def projTest' {B : Bool → Type} (s : B true)
   : B (L4L.castHEq ...
                  --^ proof of `HEq [type of rec app] Bool`
           (@K.rec (fun _ => S) (S.mk true true) k).2) :=
  L4L.castHEq ... s
             --^ very large proof of `HEq (B true) (B [above cast])`
```

- An inherent translation issue, cannot be avoided with optimizations
- May be possible to alleviate somewhat by pre-processing input to avoid unnecessary uses of PI

Scaling difficulties: dependent constant expansion

There are also cases where parts of dependent constants can make their way into the final translation output. For example:

```
inductive E where | a : E | b : E
inductive T : Nat → Nat → Prop where | mk : (n : Nat) → T n n
-- T.rec.{u} : {n : Nat} → {M : (m : Nat) → T n m → Sort u} →
--   M n (T.mk n) → {m : Nat} → (t : T n m) → M m t
macro x : Nat := ... -- some very large, long-to-typecheck term
abbrev C := fun m => (n : Nat) → T n (n + m)
def f (m : Nat) (c : C m) : E → E
| .a => @T.rec x (fun _ _ => E) .a (x + m) (c x)
| .b => .b
def fp (c : C 0) : f 0 c .a = .a := rfl -- does not reference `x`

def fpTrans1 (c : C 0) : f 0 c .a = .a :=
  L4L.castHEq
    (L4L.appArgHEq' (Eq (f 0 c .a))
      -- proof that `f 0 c .a = .a`
      (L4L.appArgHEq' (@T.rec _ (fun _ _ => E) .a _)
        (L4L.prfIrrelHEq (c x) (.mk x))))
    rfl                                  --^ references `x`
```

Scaling difficulties: dependent constant expansion

In general, subterms from expanded dependencies can accumulate, leading to poor scaling. Could be avoided using auxiliary auto-generated lemmas:

```
def T.rec_aux {n : Nat}
  {M : (m : Nat) → T n m → Sort u}
    (mtv : M n (T.mk n)) {m : Nat} (t : T n m) (p_nm : n = m) :
      HEq (@T.rec n M mtv m t) mtv
  := ...

def f_aux (m : Nat) (c : C m) (e : E)
   (p_m : m = 0) (p_e : e = E.a) : HEq (f m c e) E.a :=
  match e with
  | .a => @T.rec_aux x (fun _ _ => E) E.a (m := x + m) (c x) (...)
  | .b => E.noConfusion p_e

def fpTrans2 (c : C 0) : f 0 c E.a = E.a :=
  L4L.castHEq
    (L4L.appArgHEq' (Eq (f 0 c E.a))
      (f_aux 0 c E.a rfl rfl))
    rfl
```

Prospects: extensionality for Lean

Lean4Less's patching 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:

$$\frac{\Gamma \Vdash^{=}_{\mathsf{e}^*} A : \mathsf{U}_\ell \qquad \Gamma \Vdash^{=}_{\mathsf{e}^*} t, u : A \qquad \Gamma \Vdash^{=}_{\mathsf{e}^*} \_ : t =_A u \text{ computable}}{\Gamma \Vdash^{=}_{\mathsf{e}^*} t \Leftrightarrow u} \text{ (RFL*)}$$

and extend Lean4Less to translate from this theory "$\text{Lean}_{\mathsf{e}^*}$".

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

Progress so far

- First release of Lean4Less is available at: `https://github.com/rish987/Lean4Less`.
- Can eliminate proof irrelevance and K-like reduction from Lean
- Outputs translated environment in the form of a set of `.olean` files that follow the input file structure
- Capable of translating Lean std library and lower-level Mathlib modules (e.g. `Mathlib.Data.Real.Basic`), though it does not yet scale to larger libraries
- (paper available soon!)

Thank you for listening!