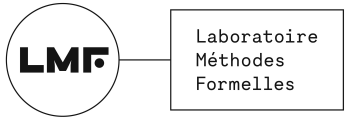
Lean4Less: A Term-Patching Framework for Eliminating Definitional Equalities in Lean (Work in Progress)

Rishikesh Vaishnav

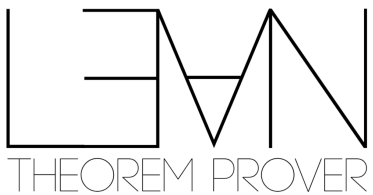
Presented at Fontainebleau Deducteam Seminar

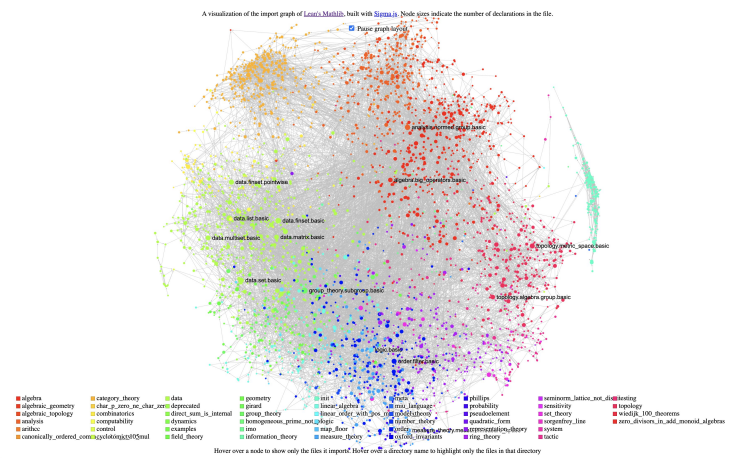
September 27, 2024



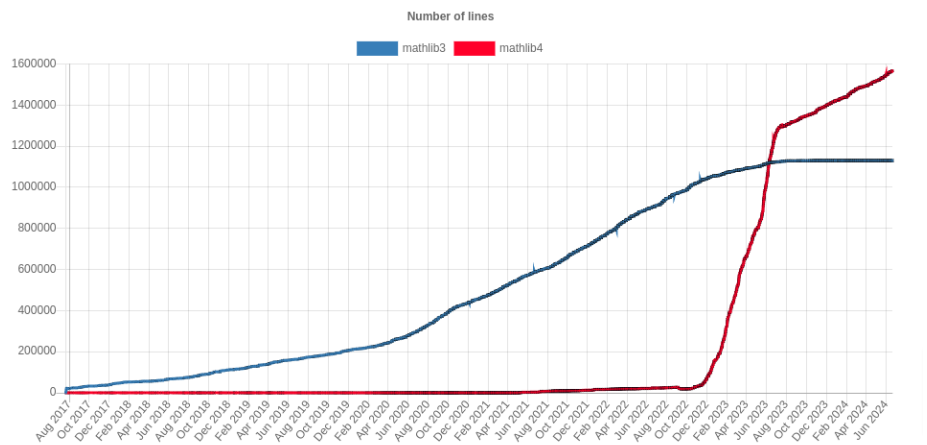
Introduction: Lean

- 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



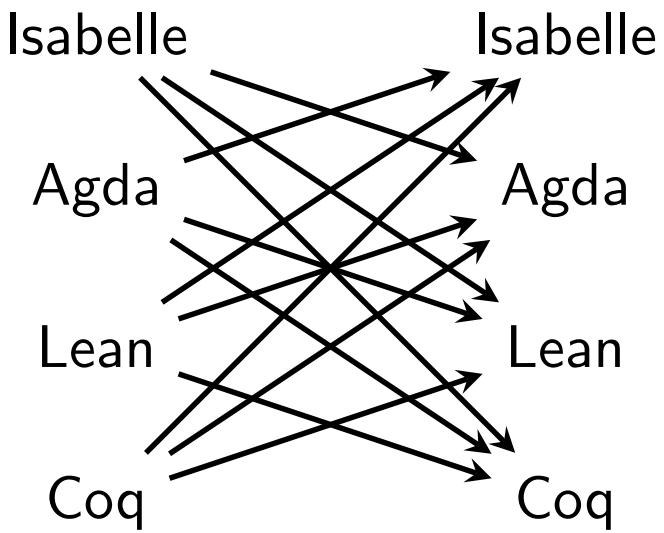
mathlib's growth

Introduction: Translation

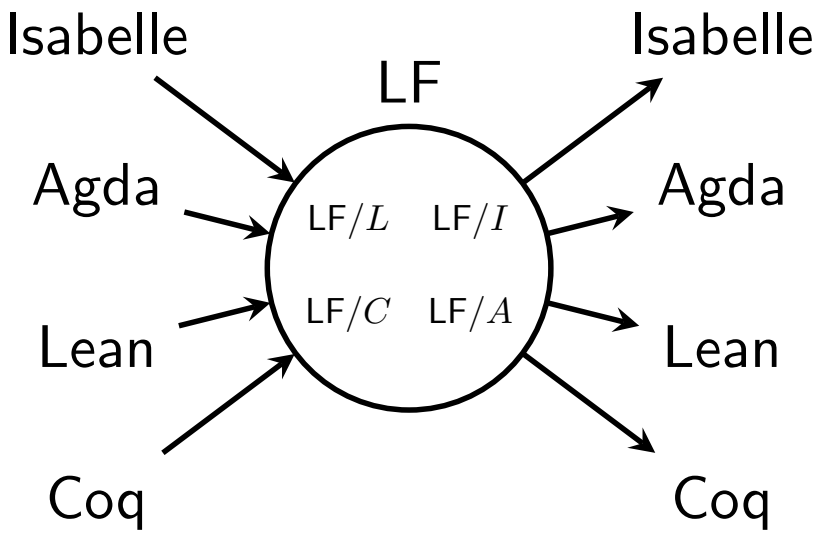
Some reasons why we would like to translate Lean to other systems:

- 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

def decrement (n : Nat) : Nat :=
  Nat.rec .z (fun n' _ => n') n

-- `decrement (Nat.s Nat.z)`
-- is defeq to `Nat.z`
example : decrement (.s .z) = .z := rfl
```

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

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`, allowing for 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

-- succeeds 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

-- fails 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 := rfl
```

```
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 (though it is not type-correct).

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 to be correctly typed, Lean must apply (PI):

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

axiom T : P → Type

-- `T p` is defeq to `T q`
-- (due to proof irrelevance)
def ex (t : T p) : T q := t
```

So, cannot directly translate this to Dedukti.

However, we can "patch" it with a PI axiom + typecasting to get around the use of (PI):

```
-- proof irrelevance, represented as an axiom
axiom prfIrrel {P : Prop} (p q : P) : p = q

theorem congrArg {A : Sort u} {B : Sort v} {x y : A}
(f : A → B) (h : x = y) : f x = f y := ...

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

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

axiom T : P → Type

def ex' (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-):

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

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

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

Extensional Type Theory and the Reflection Rule

Suppose we add the "reflection" rule of extensional type theory:

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

In this theory "Lean$_{\mathsf{e}}^{-}$", we can recover (PI) from (PI-):

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

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).

Translations can get complex when proofs appear in dependent types:

```
-- need heterogeneous equality
inductive HEq : {A : Sort u} → A →
    {B : Sort u} → B → Prop where
  | 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' := ...

-- must now be heterogeneous
axiom prfIrrel {P Q : Prop} (h : P = Q)
  (p : P) (q : Q) : HEq p q
```

```
axiom P : Prop
axiom Q : P → Prop
axiom p : P
axiom q : P
axiom Qp : Q p
axiom Qq : Q q

axiom T : (p : P) → Q p → Prop

-- with proof irrelevance, `t` would suffice
def ex (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 binder 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) := ...
```

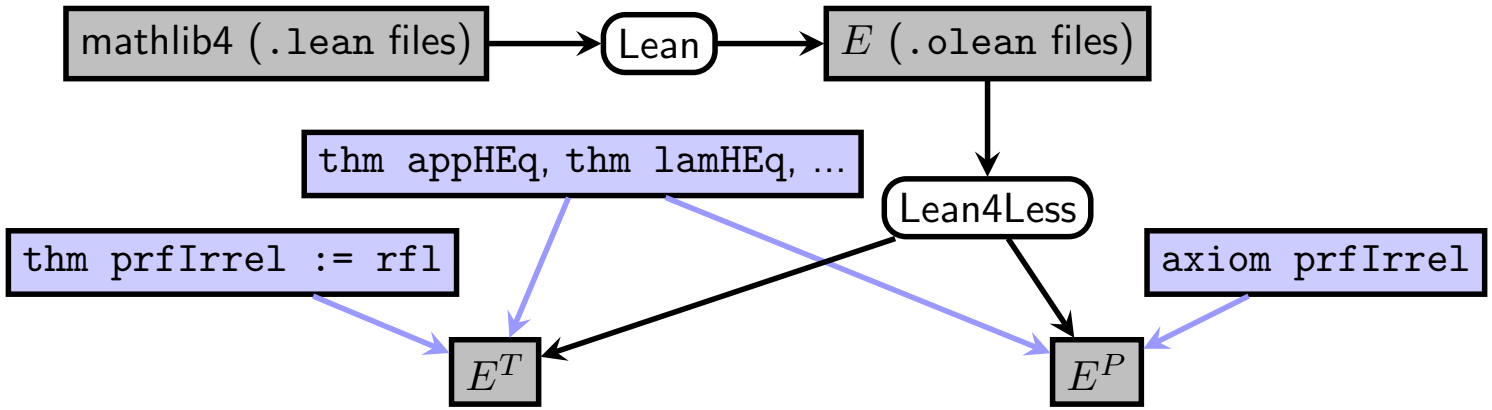
```
-- lambda congruence
theorem lamHEq {A B : Type u}
{U : A → Type v} {V : B → Type v}
  (f : (a : A) → U a) (g : (b : B) → V b)
  (hAB : A = B) (h : (a : A) → (b : B)
  → HEq a b → HEq (f a) (g b))
  : HEq (fun a => f a) (fun b => g b) := ...

-- 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

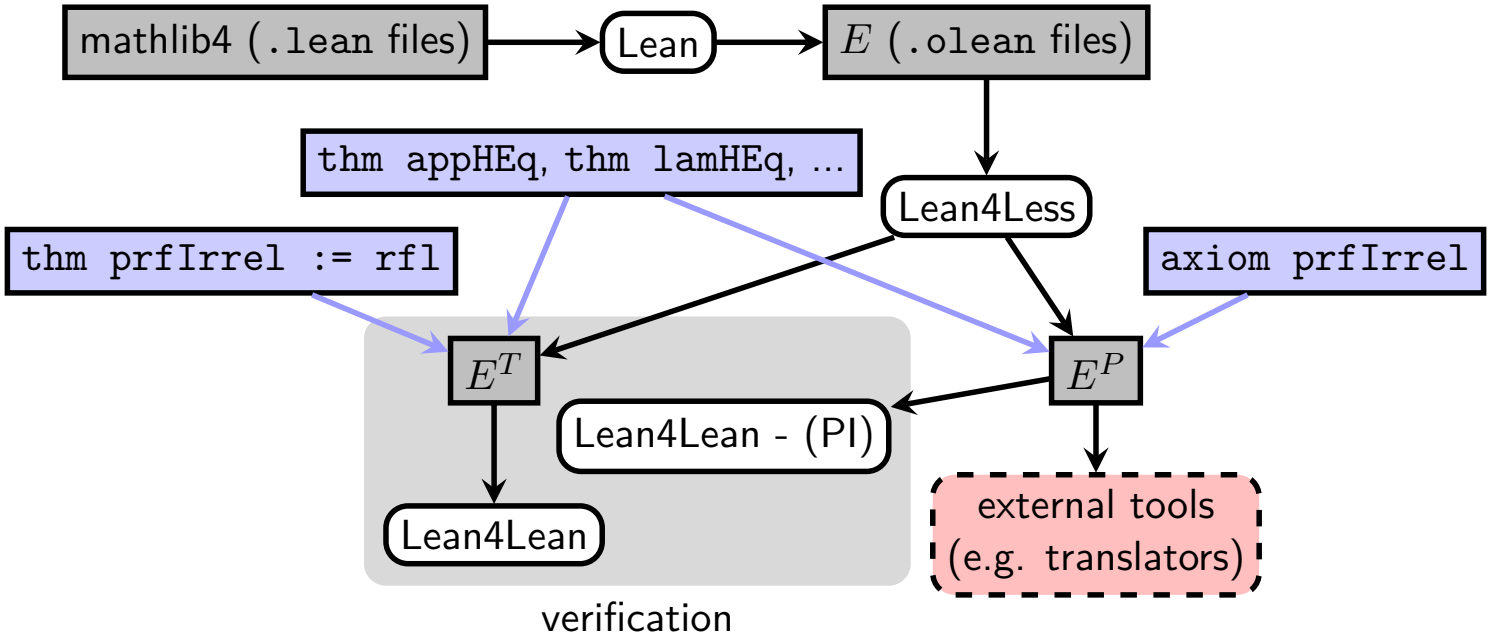
Once Lean4Less is implemented, we will test it on mathlib4:



- 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).

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 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 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 "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

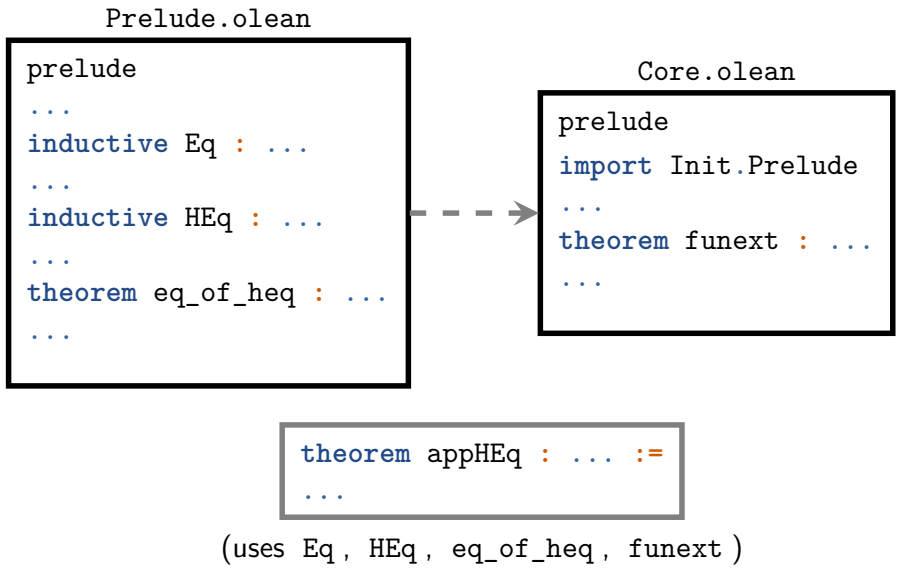
- Completed base implementation without the use of partial/unsafe definitions (ensuring termination)
- Adapted Lean4Lean to integrate it with Lean4Less as a dependency (for verification and in some parts of typechecking)
- Can now translate all of Lean's standard library and typecheck the output in Lean$^-$ (502 definitions using proof irrelevance and 126 using K-like reduction)
- Implemented a number of output optimizations to keep the output size reasonable (though can still "explode" in certain cases)

Once we can patch & typecheck mathlib with Lean4Lean - (PI), can "push the limits" by eliminating other defeqs (e.g. struct eta) to identify trickier bugs and have more confidence in the translation framework as a whole.

To do: 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.

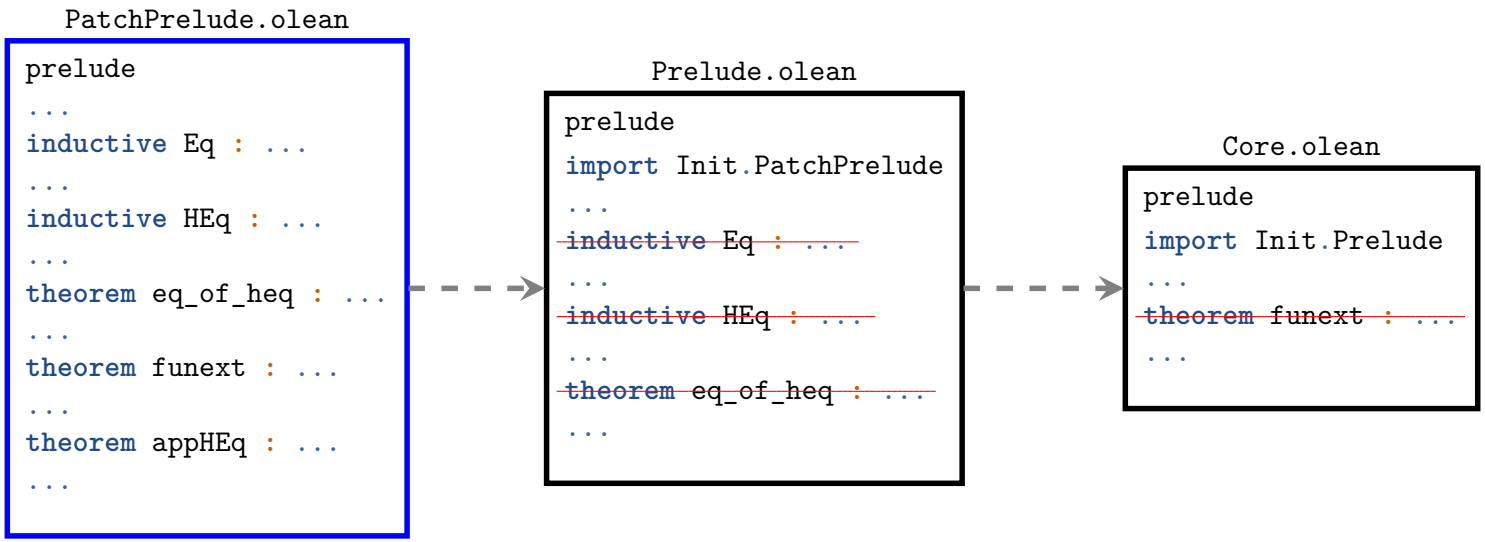
- Ideally, we would like to output separate `.olean` files using the same file structure from the input.
- However, patching lemmas depend on definitions strewn throughout the standard library:



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

To do: 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:



To do: patching lemma bootstrapping

Some issues arise when attempting to patch the patching lemmas themselves:

- Proofs require uniqueness of identity proofs (UIP) to hold definitionally, which do thanks to PI/K-like reduction
- However, in Lean$^-$, UIP only holds *propositionally*, so patching lemmas must themselves be patched
- Opens the possibility for dependency cycles, where patched patching lemmas can refer to themselves/each other
- Possible solution: manually patch patching lemmas prior to translation, using simpler lemma variants to avoid cycles
- Better solution: optimize output to extent that dependency cycles are automatically eliminated (assuming a certain declaration order)

Optimization: Minimal patching lemma variants

To address the above bootstrapping issue, it may help to use variants of patching lemmas with fewer hypotheses.

- 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$^-$

- Simpler variants may use fewer defeqs, making cycles less likely
- Also help avoid redundant reflexivity proofs in the output

Optimization: Minimal patching lemma variants

A particular lemma that uses UIP (via K-like reduction) is eq_of_heq:

```
theorem eq_of_heq {A : Sort u} {a a' : A} (h : HEq a a') : Eq a a' :=
  have (A B : Sort u) (a : A) (b : B) (h  : HEq a b)
    : (h : Eq A B) → Eq (cast h a) b :=
    h.rec (fun _ => rfl)
  this A A a a' h rfl
```

which, with a not-fully-optimized patching implementation, translates to:

```
theorem eq_of_heq {A : Sort u} {a a' : A} (h : HEq a a') : Eq a a'
  let_fun this := fun (A B : Sort u) (a : A) (b : B) (h  : HEq a b) =>
    HEq.rec (L4L.castHEq
        (L4L.forallHEq' fun (h : Eq A A) => L4L.appArgHEq (Eq (cast h a))
          (L4L.appArgHEq (Eq.rec a) h rfl (L4L.prfIrrel h rfl)))
        fun _ => rfl) h ;
  this A A a a' h rfl
```

- This results in a cycle since L4L.castHEq uses eq_of_heq
- Can be avoided by further optimizing output to use Eq instead of HEq when possible (+ defining Eq-based variants of lemmas used above), allowing a normal cast to be used instead

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.

Optimization: Application argument abstraction

This optimization is more complex than it may seem at first glance.

- Because of dependent types, later arguments may need to be abstracted even if they are Lean$^-$-defeq. For example:

```
inductive I : Type where
| left  : P → I
| right : P → I
axiom B : (i : I) → Nat → Nat → Nat → IT i → Nat → Nat → Nat → Prop
axiom Bq : B (.left q) 0 0 0 () 0 0 0
theorem absDemoB : B (.left p) 0 0 0 () 0 0 0 := Bq
```

```
def IT : I → Type
| .left _  => Unit
| .right _ => Bool
```

translates to[6]:

```
theorem absDemoB : B (I.left p) 0 0 0 Unit.unit 0 0 0 :=
  L4L.castHEq (L4L.appFunHEq Unit.unit
    (L4L.appArgHEq' (fun (i : I) (a : IT i) => B i 0 0 0 a 0 0 0)
      (I.left q) (I.left p)
      (L4L.appArgHEq I.left (L4L.prfIrrel P q p))))
  Bq
```

[6] Note: Here, we can further optimize the abstraction to obtain `fun (x : P) => B (I.left x) 0 0 0 () 0 0 0`; the definition of `IT.left` cannot depend on the proof argument b/c of weak elimination (related to proof irrelevance). However, in the general extensional case, the extra abstraction is necessary.

Optimization: Application argument abstraction

We also need to consider the case of functions with dependent arity:

```
def ITC : I → Type
| .left _  => Nat → Nat → Nat → Prop
| .right _ => Bool

axiom C : (i : I) → Nat → Nat → Nat → ITC i
axiom Cq : C (.left q) 0 0 0 0 0 0

theorem absDemoC : C (.left p) 0 0 0 0 0 0 := Cq
```

Need an extra abstraction on partial application `C (I.left p) 0 0 0`

(because `fun (i : I) => C i 0 0 0 0 0 0` is ill-typed):

```
theorem absDemoC : C (I.left p) 0 0 0 0 0 0 :=
L4L.castHEq
  (L4L.appArgHEq (fun (f : Nat → Nat → Nat → Prop) => f 0 0 0)
    (L4L.appArgHEq' (fun (i : I) => C i 0 0 0) (I.left q) (I.left p)
      (L4L.appArgHEq I.left (L4L.prfIrrel P q p))))
  Cq
```

Optimization: Application argument abstraction

Some part of the dependent argument range may depend on a previously abstracted argument, as in this example:

```
axiom Q : P → Prop
axiom Qp : Q p
axiom Qq : Q q
axiom D : (i : I) → Nat → Nat → Nat → ITD i
axiom Dq : D (.left q) 0 0 0 0 Qq 0
theorem absDemoD : D (.left p) 0 0 0 0 Qp 0 := Dq
```

```
def ITD : I → Type
| .left x  => Nat → Q x → Nat → Prop
| .right _ => Bool
```

This leads to a more complex translation, where part of the abstracted partial application's function type must also be abstracted:

```
theorem absDemoD: D (I.left p) 0 0 0 0 Qp 0 :=
L4L.castHEq
  (L4L.appHEqAB (L4L.appArgHEq Q (L4L.prfIrrel P q p))
    (L4L.appHEqABUV
      (L4L.appArgHEq (fun (aT : Prop) => Nat → aT → Nat → Prop) (L4L.appArgHEq Q (L4L.prfIrrel P q p)))
      (L4L.appArgHEq (fun (aT : Prop) => aT → Prop) (L4L.appArgHEq Q (L4L.prfIrrel P q p)))
      (L4L.appArgHEq' (fun (aT : Prop) (f : Nat → aT → Nat → Prop) (a : aT) => f 0 a 0) (Q q) (Q p)
        (L4L.appArgHEq Q (L4L.prfIrrel P q p)))
      (L4L.appArgHEq' (fun (i : I) => D i 0 0 0) (I.left q) (I.left p)
        (L4L.appArgHEq I.left (L4L.prfIrrel P q p))))
    (L4L.prfIrrelHEq (Q q) (Q p) (L4L.appArgHEq Q (L4L.prfIrrel P q p)) Qq Qp))
  Dq
```

(note the abstracted `aT` in the type of `f`).

Next steps

Within the coming weeks, I plan to:

- Fix the bootstrapping issue by optimizing the output to use `Eq` instead of `HEq` wherever possible
- Do some additional output optimizations relating to application abstraction and the reduction of any redexes produced by patching
- Attempt to translate all of mathlib
- Start work on paper describing implementation and output optimizations, and comparing runtime and output size depending on which optimizations are applied and which defeqs are eliminated
- Work on runtime optimizations

Thank you for listening!