Lean4Less: Eliminating Definitional Equalities from Lean via an Extensional-to-Intensional Translation

Rishikesh Vaishnav

Université Paris-Saclay, INRIA project Deducteam, Laboratoire Méthodes Formelles, ENS Paris-Saclay, France

Abstract. The Lean proof assistant features a typechecker kernel that makes use of a set of "definitional equalities" for identifying terms under certain syntactic and typing conditions. While providing for convenient formalization, some definitional equalities in particular complicate meta-theoretical analyses and the export of Lean proofs to other proof assistants via logical frameworks such as Dedukti. In this paper, we describe a translation from Lean to a smaller theory "Lean$^-$" with fewer such definitional equalities, specifically eliminating uses of proof irrelevance and "K-like reduction" in the typing of Lean terms. We adapt a general translation from extensional to intensional type theory, making Lean's implicit use of these definitional equalities explicit through the use of type casts and a corresponding proof irrelevance axiom. The translation has been implemented in Lean itself in a tool called Lean4Less[1], which is able to successfully translate certain libraries (e.g. the Lean standard library) to Lean$^-$. The methods developed for this translation may also be transferrable to other proof assistants based on dependent type theory.

Keywords: Lean · Dedukti · logical frameworks · extensional type theory · proof system interoperability · proof translation · dependent type theory · proof assistants

1 Introduction

Lean [11] is a proof assistant developed by the Lean FRO with type-theoretic foundations that are based on the Calculus of Inductive Constructions [13], sharing many similarities with the proof assistant Rocq [14]. It has become especially popular with mathematicians in recent years, being well-known for its "Mathlib" library [19,18], a large and quickly growing body of mathematics formalized in Lean. Lean features a small, fast kernel that attempts to be a "minimal" foundation for the sound typechecking of Lean proofs. Given Lean's popularity, it is of high interest to export Lean proofs to other proof assistants (e.g. Rocq) in order to both allow for more confidence in their correctness by typechecking them with a separate kernel, and to provide other proof assistant communities with access to various formalizations developed in Lean.

[1] `https://github.com/rish987/Lean4Less`

However, this task is complicated by certain meta-theoretical aspects of Lean. Lean's kernel, while small, is not entirely minimal, as it enforces a number of additional definitional equalities[2], such as those of proof irrelevance and "K-like reduction" (and more recently, "struct eta" and "struct-like reduction"). Such equalities are not necessarily present in other proof assistants, so special consideration must be made during translation to ensure compatibility of these features at the kernel level. One promising approach to this may be to "eliminate" them entirely, namely by performing a "pre-translation" step on well-typed terms in the original theory so that they are able to type in a strictly smaller theory (possibly extended with some axioms). It is this approach of pre-translation to eliminate definitional equalities (prior to final proof export) that we have implemented with our tool "Lean4Less", which we describe in this paper.

The remainder of the paper is structured as follows: we start by describing proof irrelevance and K-like reduction, and bring up some meta-theoretical difficulties arising from their use in Lean's typing. This motivates the translation to our target theory of "Lean$^-$", where these definitional equalities have been eliminated. In Section 2, we describe the theory behind our approach, noting that this translation task can be interpreted as a special case of a translation from extensional to intensional type theory. In Section 3, we provide an overview of how we have implemented our tool as a modification of Lean4Lean [6], an external typechecker for Lean implemented in Lean. In Section 4, we provide more details on the implementation of the translation. In Section 5, we describe some translation results on specific libraries, providing data regarding translation overhead and runtime. We conclude by discussing future directions of our work, relating in particular to the possible addition of extensional typechecking to Lean and simplifications in the analyses of certain meta-theoretical properties.

1.1 Proof Irrelevance

Lean's type theory features a definitional equality known as "proof irrelevance", which enables it to ignore the computational content of proofs when typechecking terms, only concerning itself with the equality of their propositional types. It is represented by the following rule[3]:

$$\frac{\Gamma \vdash P : \texttt{Prop}^4 \quad \Gamma \vdash h : P \quad \Gamma \vdash h' : P}{\Gamma \vdash h \equiv h'} \text{ [PI]}$$

where $\Gamma \vdash t : T$ and $\Gamma \vdash t \equiv s$ denote Lean's typing and definitional equality judgments.

[2] In type theory, "definitional equalities" are rules describing "built-in" notions of equality between terms that are implemented by the typechecker kernel, often as a formalization convenience that allows the kernel to identify a larger class of terms without requiring explicit proof from the user.

[3] The full set of typing rules in Lean was first described by Carneiro [5].

[4] Lean features a universe hierarchy of "sorts", with `Sort 0`, a.k.a. `Prop`, being the bottommost universe of propositional types. Sort typing follows the relation `Sort u : Sort (u + 1)`.

Proof irrelevance is useful, for example, in establishing the definitional equality of predicate subtype instances with equal values, but differing membership proofs. Subtypes in Lean can be defined as a parametric inductive type:

```
-- subtype inductive type parameterized by type `A` and predicate `p`
-- (curly brackets `{...}` denote auto-inferred implicit arguments)
inductive Subtype {A : Type} (p : A → Prop) where
-- we construct an instance of `Subtype A` with the constructor `mk`,
-- which takes a value `val` and proof `prf` that `val` satisfies `p`
| mk : (val : A) → (prf : p val) : Subtype p
```

Suppose that we define a subtype for natural numbers less than five:

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

Now, suppose we have two different proofs `p1 p2 : 3 < 5`. Proof irrelevance gives us a definitional equality between `NatLT5.mk 3 p1` and `NatLT5.mk 3 p2`, as one would expect, since when we consider the equality of these subtype constructions, all that we care about is the equality of their underlying values.

Forms of proof irrelevance are supported in a number of other proof assistants. Until recently, the use of proof irrelevance in Rocq had to be made explicit with an axiom[5]. However, optional support for definitional proof irrelevance has recently been added with the `SProp` type[6]. Agda supports user-annotated irrelevant function arguments and struct fields[7], in addition to a proof irrelevant type universe `Prop` (analogous to Rocq's `SProp`). F* erases the details of SMT solver-generated equality proofs [17]. The PVS proof assistant[8] features a special case of proof irrelevance in identifying predicate subtype constructions.

1.2 K-Like Reduction

Lean also features a related definitional equality rule known as "K-like reduction". It is based on the characterization of so-called "K-like" inductive types in Lean, which are defined as inductive types that that live in `Prop` and have a single constructor without any (non-parametric) arguments. Lean's equality inductive type is an example of such a K-like inductive type:

```
-- (we use `u` and `v` for level variables in universe-polymorphic
-- constants; Lean auto-infers their values when the constant is used)
inductive Eq {A : Sort u} (a : A) : A → Prop where
| refl : Eq a a
```

[5] `https://rocq-prover.org/doc/V9.0.0/stdlib/Stdlib.Logic.ProofIrrelevance.html`
[6] See `https://rocq-prover.org/doc/V9.0.0/refman/addendum/sprop.html`.
[7] See `https://agda.readthedocs.io/en/v2.5.4/language/irrelevance.html`.
[8] `https://pvs.csl.sri.com/`

We will use the notation `a = b` for the equality type construction `Eq a b`. This inductive type has two parameters, found to the left of the colon in the inductive type signature: the polymorphic type `A` and the left hand-side element `a : A`. It also has an "index", which is a special kind of inductive type parameter that is determined by the constructor: in this case, this is the right hand-side element, which, as expressed in the output type of `Eq.refl`, must be the same as (i.e. definitionally equal to[9]) the left hand-side element. In the particular case of K-like inductive types, where constructors have no arguments, indices are effectively a function of the parameters.

In general, suppose we have a K-like type `K` with n parameters, m indices and the unique constructor `mk`. We can express K-like reduction as the rule:

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

For example, the above rule applies to the equality type with $n = 2$, $m = 1$, `K` $=$ `Eq`, `mk` $=$ `Eq.refl`, $p_1 =$ `Nat`, $p_2 = 0$, and $i_1 = 0$, allowing any term `t : 0 = 0` to be reduced[10] to `Eq.refl Nat 0`. Note that K-like reduction is related to proof irrelevance, since t and `mk` $p_1\ \ldots\ p_n$ are already definitionally equal in Lean by [PI] (as K-like inductive types must live in `Prop`). However, it represents a directed equality (i.e. "rewrite rule"), rather than an undirected one, which enables a more powerful elimination principle. For example, consider the K-like type `T` below. Lean automatically generates a reduction rule for `T.rec`, the recursor (a.k.a. eliminator[11]) of `T`:

```
inductive T : Prop where | mk : T
#check (T.rec : {m : T → Sort u} → m T.mk → (t : T) → m t)
-- (`rfl` is shorthand for `Eq.refl _`, where `_` is inferred;
-- it can be used as a quick check for whether the kernel considers
-- two terms to be definitionally equal)
example : T.rec true T.mk = true := rfl
```

Normally, such reductions are limited to explicit well-typed constructions in the recursor's "major premise" argument that is eliminated upon (the `T.mk` argument above). However, K-like reduction also gives us the definitional equality:

```
example (t : T) : T.rec true t = true := rfl
```

That is, we are able to reduce on any well-typed major premise argument, without needing an explicit construction – here, it is simply the variable `t`. When

[9] By type conversion and application congruence of definitional equality.

[10] With respect to a practical typechecker implementation, during reduction we must ensure that `t` is not already an application of `Eq.refl` before applying [KLR] in order to avoid non-termination.

[11] In proof assistants, recursors/eliminators are functions that enable the definition of general transformations on inductive types; they perform a function similar to that of "pattern matching" in many functional programming languages.

reducing the recursor application, the kernel is able to "rewrite" `t` to `T.mk` using [KLR], allowing the left hand-side recursor application to reduce.

K-like reduction, in combination with Lean's impredicative `Prop` universe, results in non-termination of reduction, as shown by Abel and Coquand [1]. While its use in Lean has proven to be quite successful, such a theoretical lack of strong normalization may be part of the reason why very few other proof assistants support it. It does however exist to a limited extent in the Rocq proof assistant, where it can be enabled with the "Definitional UIP" flag[12] (this is not enabled by default to preserve certain theoretical properties, e.g. normalization).

1.3 Meta-Theoretic Challenges

While proof irrelevance, K-like reduction, and other definitional equalities in Lean are crucial conveniences in scaling mathematical formalizations, they present difficulties at the meta-theoretic level, particularly when we want to reason about or perform transformations on Lean terms based on their typing derivations. Such definitional equalities also complicate the task of exporting Lean proofs to other proof assistants, which is important to enable greater proof system interoperability and avoid duplication of work in formalizing mathematical results. Existing work translating Lean to other proof assistants, such as that of Gilbert in translating Lean to Rocq,[13] rely on the presence of similar features in the target theory. When such features are not present, direct translation becomes more difficult. We may instead look into first translating to a more universal "intermediate theory" from which we can export to several different theories.

In light of this, one promising target for proof export is Dedukti [4], a logical framework featuring dependent types and rewrite rules to ease the translation of proofs between proof assistants by translating between various encodings of different type theories. Dedukti uses the $\lambda\Pi$-calculus modulo rewriting type theory, which is intentionally designed to be as "minimal" as possible to make it a good candidate for exporting proofs between different proof assistants with various different type theories. In particular, it does not feature proof irrelevance or K-like reduction. While proof irrelevance can be encoded in Dedukti in certain special cases, as was done for the case of predicate subtyping in the proof assistant PVS [10], an encoding of the general case of proof irrelevance within Dedukti may not be possible without certain non-trival extensions to its underlying theory.

Considering the above difficulties, one may wonder whether or not it is possible to "eliminate" certain (particularly problematic) definitional equalities to some extent, by translating Lean terms to typecheck in some smaller theory that does not use them. In some cases, terms may use certain definitional equalities in "non-essential" ways, and can be rewritten in such a way as to avoid them. However, there are cases where their use *is* essential in typing, enabling proofs that

[12] `https://rocq-prover.org/doc/V8.18.0/refman/addendum/sprop.html#definitional-uip`.

[13] See `https://github.com/SkySkimmer/rocq/tree/lean-import`.

would not otherwise be possible. So, instead of eliminating definitional equalities entirely, we would like to retain them to some extent in our target theory, demoted to axiomatized/provable propositional equalities that are added to the typechecking environment, and translating terms to explicitly make use of them as needed to become typeable in the smaller theory.

2 Theoretical Background

2.1 Target Theory: Lean$^-$

To this end, we propose our target theory Lean$^-$, removing [PI] and [KLR] from our theory and adding the axiom:

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

Using this axiom, we can also represent K-like reduction, which becomes a provable proposition in this smaller theory. For instance, in the case of `T`:

```
theorem T.KLR (t : T) : T.rec true t = true :=
  -- proof that `T.rec true t = T.rec true T.mk`
  -- (without [KLR], the LHS cannot reduce to the RHS value of `true`)
  @congrArg _ _ t T.mk (T.rec true) (prfIrrel t T.mk)
```

To effect this translation, we can "inject" type casts (a.k.a. transports) around subterms in order for them to have the expected type that is imposed by a user-provided type annotation or the typing constraints of the surrounding term. For example, we can eliminate proof irrelevance when it is used directly:

```
variable (P : Prop) (p q : P) (T : P → Type)
-- `T p` is defeq to `T q` (due to proof irrelevance)
def ex (t : T p) : T q := t
theorem congrArg {A : Sort u} {B : Sort v} {x y : A}
  (f : A → B) (h : x = y) : f x = f y := ...
-- explicitly converts a term from type `A` to provably equal type `B`
def cast {A B : Sort u} (h : A = B) (a : A) : B := ...
def exTrans (t : T p) : T q := cast (congrArg T (prfIrrel p q)) t
```

and also when it is used indirectly via K-like reduction (as shown in `T.KLR`).

The Soundness Property In our target theory Lean$^-$, we have removed [PI] and [KLR] from the type theory. We would like to define some translation $|\cdot|$ on Lean-typeable terms that satisfies a soundness criterion:

$$\Gamma \vdash t : A \implies (\texttt{prfIrrel} : \forall (P : \texttt{Prop}),\ (p, q : P).\ p = q) :: |\Gamma| \vdash^{-} |t| : |A|.$$

where $\Gamma \vdash^{-} t : T$ is the notation for Lean$^-$'s typing judgment, and $|\Gamma|$ applies the translation to every type in context Γ. In words, we want the translation of

a well-typed Lean term to be well-typed in Lean$^-$ as the translation of its type, provided a proof irrelevance axiom in the typing context.

However, this property alone is not sufficient for our purposes. We would also like to ensure that type semantics are preserved by our translation. For instance, a translation that translates all types to the Lean proposition `True` and all terms to the constructor `True.intro` would be able to satisfy the above property. In particular, we would like to ensure that all translated terms are the same as the originals except possibly also having been "decorated" with type casts. We can capture this notion with the similarity relation "$\sim$" defined by Winterhalter et. al. [20]. Specifically, we want our translation to satisfy the property that, for all Lean-typeable t, we have $t \sim |t|$. This also allows translated terms to easily be translated back to the original theory by simply removing the type casts.

2.2 A Middle-Ground Extensional Theory: Lean^-_e

The above suggests that translating from Lean to Lean$^-$ may be feasible using type casting. It is reminiscent of what one may do in Lean to align types that are provably, but not definitionally equal:

```
-- addition matches on the second operand, so this is not definitional
theorem addOneComm (n : Nat) : Nat.succ n = 1 + n := ...
inductive Vec : Nat → Type where
| nil : Vec 0
| cons {n : Nat} (v : Vec n) (x : Nat) : Vec (Nat.succ n)
def vecAppend1 (n : Nat) (v : Vec n) : Vec (1 + n) :=
  -- `v.cons 1` has type `Vec (Nat.succ n)`, not `Vec (1 + n)`
  cast (congr rfl (addOneComm n)) (v.cons 1)
```

The term `v.cons 1` has the inferred type `Vec (n + 1)`, which doesn't match the annotated expected type `Vec (1 + n)` (Lean's `Nat.add` function recurses on the second argument), so we have to apply a `cast` around it using an equality proof between these types, quite similarly to what we did in `exTrans` above. This may make us question whether our task is a special case of a translation from a more general theory. If Lean were to treat the equality of `addOneComm` as definitional in the same way that it does `prfIrrel`, we would not need to wrap `v.cons 1` in a cast. It could do so if we were to, for instance, add a rule that allows *all* propositional equalities to be promoted into definitional ones. This is exactly the rule of "equality reflection" from extensional type theory (ETT), which allows *any* provable propositional equality to be considered definitional[14].

[14] However, it should be noted that this comes at the cost of rendering typechecking undecidable – for instance, it is possible to encode the halting problem as a propositional equality, which we cannot hope to decide during typechecking. For this reason, practical systems employing extensionality such as Andromeda [3], F* [17], and Nuprl [2] restrict [RFL] to some subset of provable propositional equalities.

To obtain such an extensional theory to translate from, we can add the "equality reflection" rule to Lean$^-$, obtaining the extensional theory "Lean^-_e":

$$\frac{\Gamma \vdash^-_e A : \texttt{Sort u} \quad \Gamma \vdash^-_e t, s : A \quad \Gamma \vdash^-_e \_ : t = s}{\Gamma \vdash^-_e t \equiv s}\ [\text{RFL}]$$

using the notation $\Gamma \vdash^-_e t : T$ and $\Gamma \vdash^-_e t \equiv s$ for Lean^-_e's typing and definitional equality judgments. A translation from Lean to Lean^-_e is simply the identity function, as we have via [RFL]:

$$\frac{\Gamma \vdash^-_e P : \texttt{Prop} \quad \Gamma \vdash^-_e p, q : P \quad \Gamma \vdash^-_e \texttt{prfIrrel}\ p\ q : p = q}{\Gamma \vdash^-_e p \equiv q}$$

which is equivalent to [PI]. We can derive a similar rule for [KLR]. In fact, because Lean^-_e's theory is extensional, Lean's typing is a strict subset of Lean^-_e's.

So, because for any t, $\Gamma \vdash t : A \implies \Gamma \vdash^-_e t : A$, we can reformulate our problem as finding a translation $|\cdot|$ to Lean$^-$ respecting soundness w.r.t. Lean^-_e:

$$\Gamma \vdash^-_e t : A \implies (\texttt{prfIrrel} : \forall (P : \texttt{Prop}),\ (p, q : P).\ p = q) :: |\Gamma| \vdash^- |t| : |A|.$$

This is an instance of the general problem of translating from extensional to intensional type theory (where any type theory lacking [RFL] is considered "intensional"). Such a translation is possible, with a formally verified implementation in Rocq by Winterhalter et. al. in `ett-to-itt` [20,21], which builds on previous work by Oury [12] and Hofmann [9], with the first result showing conservativity of ETT over ITT demonstrated by Hofmann [8].

This translation places certain restrictions on the target intensional theory, namely that it exhibits propositional uniqueness of identity proofs (UIP) and function extensionality. Lean$^-$ satisfies UIP thanks to `prfIrrel`:

```
theorem UIP {A : Sort u} (x y : A) (p q : x = y) : p = q := prfIrrel p q
```

Lean$^-$ also satisfies function extensionality with the theorem `funext` from the Lean standard library, where it is proven through the use of quotient types:

```
-- (module `Init.Core`)
theorem funext {A : Sort u} {B : A → Sort v} {f g : (x : A) → B x}
    (h : (x : A) → f x = g x) : f = g := ...
```

Restrictions are also placed on the source extensional theory by requiring an ETT syntax with domain- and codomain-annotated lambda and application constructors, which Lean does not have. We skirt this requirement through the use of an extra "`hUV`" premise in our application congruence lemma (see Section 4.1).

Our theories can be summarized in the following table:

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

Practically speaking, our translation does not need to implement a full ETT-to-ITT translation. We only care about translating terms that are already typeable in Lean, so proof irrelevance is the only definitional equality we need to make explicit. Nevertheless, this does not afford us any real simplifications in the translation algorithm. Proof irrelevance may be used during typechecking to the same extent as general extensional equalities, as there are no syntactic restrictions on where proofs can appear in terms. In particular, they can appear within types, leading to some fairly complex translations (see Appendix A).

Such a translation must generalize the equality type `Eq` to the heterogeneous equality type `HEq`, which is able to take different left- and right-hand side types:

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

We use the notation `a == b` for `HEq a b`. This is a different formulation of heterogeneous equality from the one used by Winterhalter et. al. in [20], where the construction also carries a proof of equality of the left- and right-hand side's types. While such a formulation makes for more convenient correctness proofs, it is less convenient for an actual implementation, so we instead choose to return to the "John Major equality" used by Oury [12], which is a more compact and equivalent formulation already defined in the Lean standard library in the `HEq` type (`JMeq` in the Rocq standard library).

3 Implementation Overview

3.1 Adapting `ett-to-itt`?

Although a Rocq-verified translation from ETT to ITT already exists in the `ett-to-itt` repository [21], which could be extracted to an executable OCaml program [7] and possibly used in our translation, there would be a number of challenges associated with this approach, largely on account of its focus on the correctness of its translation, rather than its practicality. In particular, the extracted code would require as input some representation of Lean typing derivations, which Lean currently provides no way to obtain. Instead, we prefer to instead take the approach of modifying an existing typechecker to construct a translation in parallel to typechecking, where we have access to the typing derivation steps implicitly from the steps taken by the typechecker in deciding the well-typedness of Lean terms.

Such an approach would allow us to handle Lean's definitional equalities on a more modular basis, being able to choose which ones we eliminate at the level of the translation itself, rather than as a post-processing step. It will also allow us to retain some runtime optimizations in the Lean kernel that could translate into output optimizations, and, using utilities offered by the typechecker such as type inference and weak head normal form computation, more easily implement some output optimizations of our own (see Section 4.3). Also, by performing our translation in parallel to typechecking, we can implement a translation that only inserts type casts where necessary for the term to be well-typed in Lean$^-$ (see Section 4.2) – effecting, in this way, a kind of "patching" typechecker.

3.2 Modifying Lean4Lean

A promising Lean kernel implementation to modify to achieve our translation is Carneiro's "Lean4Lean" [6], a port of Lean's C++ kernel typechecker code into Lean, with the beginnings of the formalization of certain meta-theoretical properties in the direction of the MetaRocq project [16]. Modifying a typechecker that is implemented in Lean itself provides us with several benefits. As Lean is a partly bootstrapped language, many of its higher-level features are implemented exclusively in Lean, which use a number of helper functions for traversing and constructing expressions, manipulating free/bound variables, modifying the typechecking environment, etc., that will be useful in our own implementation. Also, Lean's orientation towards formal proof and typechecking afford us certain "soft" guarantees in the correctness of our implementation, and leaves the door open to an eventually fully verified translation on account of Lean's capabilities as a general theorem prover.

Lean4Lean's typechecker implements a bidirectional typechecking algorithm using three primary mutually recursive functions (found in `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 := ...
```

- `inferType` is a type inference function that checks that `e` is well-typed (throwing an error if it is not), returning its inferred type.
- `isDefEq` returns whether or not the well-typed terms `t` and `s` are definitionally equal according to Lean's definitional equality judgment.
- `whnf` reduces an expression to its weak-head normal form (WHNF). It is a subroutine of `isDefEq`, where terms must sometimes be (partly) reduced to determine if they are definitinally equal.

In "Lean4Less", our translation implementation adapted from Lean4Lean, we modify the return values of these functions as follows:

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

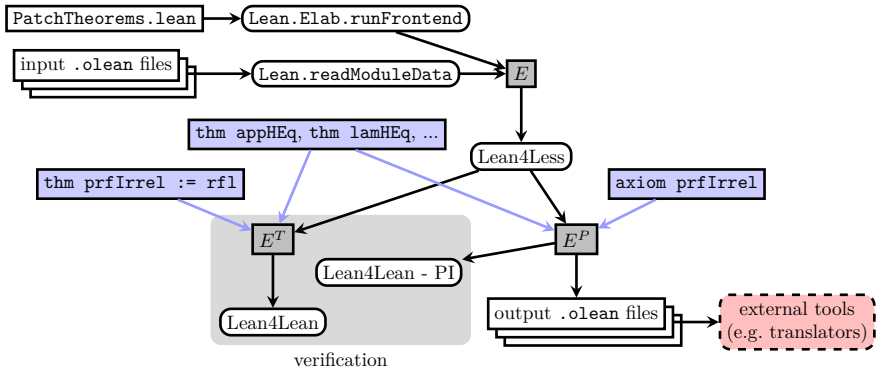
(`PExpr` and `EExpr` are Lean4Less-specific types for representing translated terms and equality proof, respectively). All three functions *optionally* return a patched expression/equality proof depending on whether proof irrelevance was used in typing/definitional equality checking. If it was not, then we return `Option.none`, indicating that no equality proof/translation was required.

The `inferType` function may now also return a translated version of the input expression injected with transports where required by typing constraints (see Section 4.2) – note that the first return value is the original inferred type return value, and we maintain that this inferred type is Lean$^-$-typeable (that is, it is a translation to Lean$^-$ of the type that would have normally been inferred by Lean4Lean's `inferType` function). The `isDefEq` function now also possibly returns a generated proof of equality between the input terms, and the `whnf` function may also return a proof of equality between the input term and its weak head normal form. Both functions return *heterogeneous* equality proofs with the type `HEq` (in particular, `whnf` must also return a heterogeneous equality proof because the type of the input term may change during reduction).

A returned proof from `isDefEq` or `whnf` can be interpreted as a "trace" of the typechecker's steps in deciding definitional equality/performing WHNF reduction. For instance, if the typechecker determines that the applications `f a` and `f b` are definitionally equal, where proof irrelevance was used at some point when comparing `a` and `b` to produce a proof term `p : a = b`, Lean4Less will construct a proof using Lean's `congrArg` lemma in order to produce the proof term `congrArg f a b p : f a = f b`.

3.3 Verification

Once we have our translated output from Lean4Less, we can verify that it is well-typed in Lean$^-$. Specifically, for some output environment E^P translated from an input environment E, we typecheck E^P using a modified fork of Lean4Lean with proof irrelevance and K-like reduction disabled. We must also verify that our translation did not change the semantics of annotated constant types as a result of translation – as explained in Section 2.1, the output of our translation should only "decorate" the input with casts between types that are already Lean-defeq. For this, we generate a verification environment E^T containing equality theorems between the original and translated types of every defined constant, proven by reflection. We then typecheck E^T with the normal Lean kernel. Our translation and verification workflow is summarized in the diagram below:



4 Implementation Details

4.1 Congruence Lemmas

In the process of translating from Lean to Lean$^-$, we use a number of specialized definitions to cast terms and build the needed type equality proofs[15]. In particular, we need a set of "congruence lemmas" to compose equality proofs from the proofs of equality of corresponding subterms, for the forall, lambda, and application cases:

```
theorem forallHEqABUV' {A B : Sort u} {U : A → Sort v} {V : B → Sort v}
  (hAB : A == B) (hUV : (a : A) → (b : B) → a == b → U a == V b)
  : ((a : A) → U a) ((b : B) → V b) := ...
theorem lambdaHEqABUV' {A B : Sort u} {U : A → Sort v} {V : B → Sort v}
  (f : (a : A) → U a) (g : (b : B) → V b)
  (hAB : A == B) (hfg : (a : A) → (b : B) → a == b → f a == g b)
  : (fun a => f a) == (fun b => g b) := ... -- (uses funext)
theorem appHEqABUV' {A B : Sort u} {U : A → Sort v} {V : B → Sort v}
  (hAB : A == B) (hUV : (a : A) → (b : B) → a == b → U a == V b)
  {f : (a : A) → U a} {g : (b : B) → V b} {a : A} {b : B}
  (hfg : f == g) (hab : a == b)
  : f a == g b := ...
```

`appHEqABUV'` contains the additional hypothesis `hUV` that allows us to equate `U` and `V` in its proof. This enables us to prove the lemma without the presence of domain- and codomain-annotated lambda and application constructors, which was a requirement on the source ETT syntax imposed by [20] in order to be able to prove a version of this lemma that does not carry this hypothesis[16]. While it may seem feasible to derive this hypothesis from the equality of the types of `f` and `g` implied by `hfg`, this is not possible in Lean without the addition of a "forall η" axiom with the signature:

```
-- (not used by our translation)
axiom forallEta : ((a : A) → U a) = ((a : A) → V a) → U = V
```

Assuming such an axiom breaks some theoretical properties of Lean, in particular its interpretation under a cardinality model where all types of equal size are considered equal[17].

[15] The full list of translation-specific constants can be found here: `https://github.com/rish987/Lean4Less/blob/main/patch/PatchTheorems.lean`

[16] For a verified translation, using this hypothesis requires a proof that it can always be inhabited, which has not been shown by Winterhalter et. al. [21]. However, we have not had any problems proving this hypothesis on-the-fly as a part of our translation.

[17] If we assume this axiom, we can show a counterexample to the cardinality model as follows: Let `A := Fin 2`, and let `U := fun x => if x = 0 then Bool else Unit` and `V := fun x => if x = 0 then Unit else Bool`. Then, we have the function type cardinalities `|(a : A) → U a| = |(a : A) → V a| = 2`, allowing us to derive `U = V` from `forallEta`. By application congruence `U 0 = V 0`, which contradicts that `|U 0| = 2` $\neq$ `|V 0| = 1`.

We also need the proof irrelevance axiom and its extension to provably equal proof types. For convenience, we also add a heterogeneous cast function:

```
axiom prfIrrel {P : Prop} (p q : P) : p = q
theorem prfIrrelHEq {P : Prop} (p q : P) : p == q := ...
theorem prfIrrelHEqPQ {P Q : Prop} (hPQ : P == Q)
   (p : P) (q : Q) : p == q := ...
def castHEq {A B : Sort u} (h : A == B) (a : A) : B :=
  cast (eq_of_heq h) a
```

These constants, along with all of their dependencies, need to be enumerated to Lean4Less to be added to the environment first, since any later definitions may reference them as a result of translation. Importantly, they must already be well-typed in Lean$^-$ and should not require translation themselves, since this could result in cyclic self-references (see Appendix C).

4.2 Producing Patched Terms

During translation, the output is obtained by "injecting" type casts into the terms around subterms whose expected and inferred types are not Lean$^-$-defeq. Expected type requirements can arise either from user-provided annotations or from typing restrictions imposed by certain typing rules. The type casts require a proof of equality between these expected and inferred types, which is computed with a call to `isDefEq`. More details on the computation of these equality proofs are provided in Appendix D.

User-provided type annotations can come from constant signatures or let bindings (a.k.a. local definitions). In the case that the annotated types do not match, we cast the entirety of the constant/let body. Checking that constant type signatures and inferred body types are equal is performed at the highest level of translation/typechecking, that is, when adding constants to the typing environment. Checking let bindings, on the other hand, occurs as a subroutine of type inference.

Type casts may also be inserted due to the following typing rules:

$$\frac{\Gamma \vdash A : \texttt{Sort u} \quad \Gamma, x : A \vdash e : B}{\Gamma, x : A \vdash \lambda x : A.\ e : \forall x : A.\ B}\ [\text{LAM}] \qquad \frac{\Gamma \vdash A : \texttt{Sort u} \quad \Gamma, x : A \vdash B : \texttt{Sort v}}{\Gamma \vdash \forall x : A.\ B : \texttt{Sort (imax u v)}}\ [\text{ALL}]$$

$$\frac{\Gamma \vdash e : \forall x : A.\ B \quad \Gamma \vdash e' : A}{\Gamma \vdash e\ e' : B[e'/x]}\ [\text{APP}] \qquad \frac{\Gamma \vdash A : \texttt{Sort u} \quad \Gamma \vdash e : A \quad \Gamma, x : A \vdash b : B}{\Gamma \vdash \texttt{let}\ (x : A)\ :=\ e\ \texttt{in}\ b : B[e/x]}\ [\text{LET}]$$

The rules [LAM], [ALL], and [LET] require the binder types (and output type, in the case of [ALL]) to be sorts, so these types may be cast if their inferred types are not Lean$^-$-defeq to some `Sort u`. Typing restrictions are also enforced by [APP], where the domain type of the function must definitionally match the inferred type of the argument, with the argument being cast if this is not the case. The function itself may also be cast, if its inferred type is not Lean$^-$-defeq to some function type.

The translation of a Lean constant is identical to the original, save for the fact that various subterms may have been "decorated" by casts (that is, they are related by the "$\sim$" similarity relation described in [20]). It is easy to recover the input Lean term from its Lean$^-$ translation: one must simply remove all type casts introduced by the translation, which are easy to identify as they use the translation-specific `castHEq` cast function.

4.3 Output/Runtime Optimizations

Output and runtime optimizations are particularly important for a tool like Lean4Less, to be able to scale up the translation to large libraries and to have a reasonably sized output that avoids redundancy. Additionally, it is important to have an efficient implementation that enables the translation to complete within a reasonable amount of time without excessive memory requirements. By virtue of being based on an efficient typechecker implementation, Lean4Less already enjoys many output and runtime optimizations that transfer over from the kernel. For instance:

- Lean uses "lazy δ-reduction" in its `isDefEq` check, avoiding the expansion of equal δ-expandable constant function application heads where possible, opting to first perform a comparison on each pair of arguments. This translates to an output optimization in which we can also avoid expanding these constants in the output when generating equality proofs.
- Lean's proof irrelevance check is placed very early on in the `isDefEq` check, ensuring that we do not needlessly compare proof subterms if we already know that the proof types are equal (thus making the proofs definitionally equal by proof irrelevance). This also becomes an output optimization, because we can immediately output an equality proof using the `prfIrrel` axiom, rather than possibly producing a larger proof resulting from a more detailed comparison of subterms (in the case that the proofs can be shown equal without applying [PI]).
- Lean's kernel makes use of a cache for recording previously computed weak-head normal forms. Lean4Less adapts this cache to store an equality proof in addition to the weak-head normal form itself, and can be queried to avoid unnecessary computations. This translates into an output optimization since these redundant proofs will also share object pointers in the `.olean` output.

Lean4Less also implements some optimizations of its own, not described here.

5 Results

We have tested our translation on the Lean standard library and various lower-level Mathlib modules, verifying our output in the manner described in Section 3.3. We have already had success in translating significant subsets of Mathlib to Lean$^-$, for instance Lean's real numbers library `Mathlib.Data.Real.Basic`,

containing several thousands of lines of code and thousands of uses of proof irrelevance and K-like reduction.

We benchmark our translation on `Std`, the Lean core standard libary, and on the mathlib library `Mathlib.Algebra.Order.Field.Rat`, with the versions of both libraries using Lean toolchain `v4.16.0-rc2`. We report below on some measures relating to the translation of these modules on a machine with an Intel Xeon 8-core CPU @ 2.20GHz and 32 GB RAM:

Module	Total Constants	Constants Using [PI]/[KLR] (% of total)	Input/Output Environment Size (Overhead)	Translation Runtime	Input/Output Typechecking Runtime (Overhead)[18]
`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%)

The standard library translation overhead of 15.5% is not very excessive relative to the 6% of total constants using proof irrelevance/K-like reduction, and we observe an even more modest translation overhead when translating an actual `Mathlib` module. In both cases, however, this is somewhat disproportionate to the amount of extra typechecking runtime overhead translation incurs. It is not clear how much of this overhead is truly unavoidable, but more work can certainly to be done to optimize the output size.

We can see above that translation takes significantly longer than typechecking, and we have found that the translation tends to get "stuck" for significant amounts of time translating certain constants, sometimes taking longer than ten minutes to translate a single definition. Such long-running translations also consume significant amounts of memory, which has proven to be a prohibitive factor in attempting to translate larger mathlib libraries. Further investigation is needed here. Such slowdowns may be related to general scaling problems that are closely tied to output inefficiencies, and may be resolved through the implementation of further output optimizations – for instance, the generation of auxiliary helper definitions and the more efficient use of caching.

6 Prospects and Conclusion

Because Lean4Less implements a special case of the ETT-to-ITT translation, an immediate interest is the possible adaptation of its translation framework for use in a general extensional-to-intensional translation. This could enable the adoption of new, possibly user-specified definitional equalities in Lean, while maintaining the ability to translate back to Lean's core type theory, producing terms that are checkable with the same small, trusted kernel. Such a development could take Lean in the direction of being an extensional proof assistant, which could significantly simplify many reasoning tasks where equality goals and hypotheses feature prominently. More details on this possible future development are provided in Appendix E.

[18] When run with the Lean and Lean$^-$ kernels, respectively (i.e. Lean4Lean with and without PI/K-like reduction)

Another potential benefit of having a translation from Lean to Lean$^-$ is that it can simplify meta-theoretical analyses of Lean's type theory by enabling us to use Lean$^-$ as a "proxy theory" for Lean itself. Specifically, some important meta-theoretical results, such as consistency, could be shown for Lean$^-$ and automatically transfer to Lean, provided the correctness of the translation implementation. More details are provided in Appendix F.

Additionally, while this work primarily concerns a particular implementation of an extensional-to-intensional translation applied specifically to eliminating the use of proof irrelevance in the typing of Lean terms, the framework developed for Lean4Less should be general enough to extend to eliminate other definitional equalities present in the Lean kernel, for instance the "struct eta" rule (and its reduction counterpart), and Lean's special reduction rules for quotient type eliminators. In addition, the techniques and optimizations developed here could be transferrable to similar translations implemented for other proof assistants, either for the purpose of proof export or for extending them to have extensional-like features of their own.

Conclusion In this paper, we describe the theory, design, and implementation of a tool that is capable of translating Lean to smaller theories through the implementation of a more general translation framework from extensional to intensional type theory. We have described how we have adapted our translation from an independent typechecker kernel implementation for Lean called "Lean4Lean" [6]. Our tool, "Lean4Less", has been successfully able to translate certain medium-sized libraries, and we hope to scale up our translation to handle larger formalizations. We believe that this work sets the foundation for the first practical translation from extensional to intensional type theory that has been implemented for a proof assistant. Such a translation may enable future extensions to the Lean kernel, allowing for more convenient mathematical formalization while retaining the ability to translate terms back to the original theory to typecheck with the same small, trusted kernel. Additionally, while this work primarily concerns a particular implementation of an extensional-to-intensional translation applied specifically to eliminating the use of proof irrelevance in the typing of Lean terms, the general techniques and optimizations developed here could be transferrable to similar translations that may be implemented for other proof assistants.

Acknowledgments. This publication is based upon work completed under COST Action EuroProofNet, CA20111, supported by COST (European Cooperation in Science and Technology).

Disclosure of Interests. The author claims no competing interests.

References

1. Abel, A., Coquand, T.: Failure of normalization in impredicative type theory with proof-irrelevant propositional equality. Logical Methods in Computer Science **Volume 16, Issue 2**, 14 (Jun 2020). https://doi.org/10.23638/LMCS-16(2:14)2020
2. Allen, S., Constable, R., Eaton, R., Kreitz, C., Lorigo, L.: The nuprl open logical environment. pp. 170–176 (12 2006). https://doi.org/10.1007/10721959_12
3. Bauer, A., Gilbert, G., Haselwarter, P.G., Pretnar, M., Stone, C.A.: Design and Implementation of the Andromeda Proof Assistant. In: Ghilezan, S., Geuvers, H., Ivetic, J. (eds.) 22nd International Conference on Types for Proofs and Programs (TYPES 2016). Leibniz International Proceedings in Informatics (LIPIcs), vol. 97, pp. 5:1–5:31. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2018). https://doi.org/10.4230/LIPIcs.TYPES.2016.5
4. Blanqui, F., Dowek, G., Grienenberger, E., Hondet, G., Thiré, F.: A modular construction of type theories. Logical Methods in Computer Science **Volume 19, Issue 1**, 12 (Feb 2023). https://doi.org/10.46298/lmcs-19(1:12)2023
5. Carneiro, M.: The Type Theory of Lean. Master's thesis (2019), https://github.com/digama0/lean-type-theory/releases/tag/v1.0
6. Carneiro, M.: Lean4lean: Towards a formalized metatheory for the lean theorem prover (2024), https://arxiv.org/abs/2403.14064
7. Forster, Y., Sozeau, M., Tabareau, N.: Verified extraction from coq to ocaml (Jun 2024). https://doi.org/10.1145/3656379
8. Hofmann, M.: Conservativity of equality reflection over intensional type theory. In: Selected Papers from the International Workshop on Types for Proofs and Programs. p. 153–164. TYPES '95, Springer-Verlag, Berlin, Heidelberg (1995)
9. Hofmann, M., Rijsbergen, C.J.: Extensional Constructs in Intensional Type Theory. Springer-Verlag, Berlin, Heidelberg (1997)
10. Hondet, G., Blanqui, F.: Encoding of Predicate Subtyping with Proof Irrelevance in the λΠ-Calculus Modulo Theory. In: de'Liguoro, U., Berardi, S., Altenkirch, T. (eds.) 26th International Conference on Types for Proofs and Programs (TYPES 2020). Leibniz International Proceedings in Informatics (LIPIcs), vol. 188, pp. 6:1–6:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2021). https://doi.org/10.4230/LIPIcs.TYPES.2020.6
11. Moura, L.d., Ullrich, S.: The lean 4 theorem prover and programming language. In: Automated Deduction – CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings. p. 625–635. Springer-Verlag, Berlin, Heidelberg (2021). https://doi.org/10.1007/978-3-030-79876-5_37
12. Oury, N.: Extensionality in the calculus of constructions. In: Proceedings of the 18th International Conference on Theorem Proving in Higher Order Logics. p. 278–293. TPHOLs'05, Springer-Verlag, Berlin, Heidelberg (2005). https://doi.org/10.1007/11541868_18
13. Paulin-Mohring, C.: Introduction to the calculus of inductive constructions. In: Paleo, B.W., Delahaye, D. (eds.) All about Proofs, Proofs for All, Studies in Logic (Mathematical logic and foundations), vol. 55. College Publications (Jan 2015), https://inria.hal.science/hal-01094195
14. Rocq Community: The rocq theorem prover, https://rocq-prover.org/
15. Selsam, D., de Moura, L.: Congruence closure in intensional type theory. CoRR **abs/1701.04391** (2017)

16. Sozeau, M., Boulier, S., Forster, Y., Tabareau, N., Winterhalter, T.: Coq coq correct! verification of type checking and erasure for coq, in coq. Proc. ACM Program. Lang. **4**(POPL) (Dec 2019). https://doi.org/10.1145/3371076
17. Swamy, N., Hriţcu, C., Keller, C., Rastogi, A., Delignat-Lavaud, A., Forest, S., Bhargavan, K., Fournet, C., Strub, P.Y., Kohlweiss, M., Zinzindohoue, J.K., Zanella-Béguelin, S.: Dependent types and multi-monadic effects in f*. SIGPLAN Not. **51**(1), 256–270 (Jan 2016). https://doi.org/10.1145/2914770.2837655
18. The mathlib community: mathlib4 (Github), https://github.com/leanprover-community/mathlib4
19. The mathlib community: The lean mathematical library. In: Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs. p. 367–381. CPP 2020, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3372885.3373824, https://doi.org/10.1145/3372885.3373824
20. Winterhalter, T., Sozeau, M., Tabareau, N.: Eliminating reflection from type theory. Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs (2019), https://api.semanticscholar.org/CorpusID:57379755
21. Winterhalter, T., Tabareau, N.: ett-to-itt (Github), https://github.com/TheoWinterhalter/ett-to-itt

A A Complex Translation

Lean4Less may produce complex translations in particular as a result of proofs appearing in dependent types, as demonstrated in the example below:

```
-- HEq version of `congrArg`
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) → a == b → U a == V b)
  (hfg : f == g) (hab : a == b)
  : f a == g b := ...
theorem eq_of_heq {A : Sort u} {a a' : A} (h : a == a') : a = a' := ...
-- proved using `prfIrrel`
theorem prfIrrelHEqPQ {P Q : Prop} (h : P = Q) (p : P) (q : Q) : p == q := ...

variable
  (P : Prop) (p q : P)
  (Q : P → Prop) (Qp : Q p) (Qq : Q q)
  (T : (p : P) → Q p → Prop)

def ex (t : T p Qp) : T q Qq := t
-- with proof irrelevance, `t` would have sufficed
def exTrans (t : T p Qp) : T q Qq := cast (eq_of_heq
  -- T p Qp == T q Qq
  (appHEq
    -- Q p = Q q
    (congrArg Q (eq_of_heq (prfIrrel p q)))
    -- Prop == Prop
    (fun _ _ _ => HEq.rfl)
    -- T p == T q
    (appHEq rfl ... HEq.rfl (prfIrrel rfl p q))
    -- Qp == Qq
    (prfIrrelHEqPQ
      -- Q p = Q q
      (congrArg Q (eq_of_heq (prfIrrel p q)))
      Qp Qq)))
  t
```

Here, we must produce a proof of equality between `T p Qp` and `T q Qq`. Equality between the partial applications `T p : Q p → Prop` and `T q : Q p → Prop` must necessarily be stated as a heterogeneous equality `T p == T q`, because their types are not Lean^{-}-defeq; we use `appHEq` to construct this proof. We next have the task of showing (heterogeneous) equality between the arguments `Qp : Q p` and `Qq : Q q`. In Lean, `Qp` and `Qq` are definitionally equal on account of proof irrelevance, since their types `Q p` and `Q q` are definitionally equal propositions, again by proof irrelevance, which lets us equate `p : P` and `q : P`. This nested use of proof irrelevance introduces some more complexity: to prove equality between `Qp` and `Qq`, we must use the heterogeneous version of the proof

irrelevance lemma, `prfIrrelHEqPQ`, that requires a proof of equality between the propositional types `Q p` and `Q q` and produces a heterogeneous equality proof.

B Problems with a Translation Extracted from `ett-to-itt`

A ETT-to-ITT translation implementation could theoretically be extracted from a constructive formal proof of conservativity of ETT over ITT, like that found in `ett-to-itt` [21]. While this would have the benefit of providing us with a verified, bug-free program implementing this translation, it would also leave us with a number of practical drawbacks:

- The translation formalized by `ett-to-itt` takes as input extensional typing derivations, rather than expression terms. Lean currently has no representation or output of typing derivations, so we would have to construct these derivations ourselves, likely by modifying a kernel implementation to produce them as "traces" of typechecking – a more efficient translation can be implemented by modifying a kernel implementation to directly output translated terms without the need for an intermediate typing derivation.
- `ett-to-itt`'s input derivations are assumed to come from a *minimal extensional theory* which is that does not contain Lean's theory. To align a Lean derivation with this minimal theory, we would have to do some preliminary alignment on the derivation to eliminate uses of Lean-specific typing rules, such as proof irrelevance, struct- and K-like reduction, quotient reduction, struct and function eta, etc. This will likely consist of replacing any uses of such rules with applications of [RFL] using the relevant lemma/axiom (as we have done above in replacing [PI] with [RFL] using the axiom `prfIrrel`).
- `ett-to-itt`'s output terms are typeable in a *minimal intensional theory*. We would correspondingly have to do some post-processing on this output in order to recover Lean-typable terms that do not use the extra axioms/lemmas introduced in the pre-processing step described above (except for the ones corresponding to definitional equalities that we are actually trying to eliminate).
- The output of `ett-to-itt` will likely be unacceptably large because it is directly derived from a formalization that makes an only very limited attempt at optimizing the output size (by eliminating redundant casts up to β-equivalence of the types being cast). Attempting some post-hoc optimizations on this large output will likely be an unwieldy task with sub-optimal outcomes.

C Bootstrapping Lemmas

The congruence lemmas shown in Section 4.1 are all proven in Lean with the usual high-level Lean tactics[19]. As elaborated, they are in fact already valid Lean$^-$ proofs – they happen to not use proof irrelevance or K-like reduction in their typing, so their translation to Lean$^-$ amounts to the indentity function with no risk of introducing cyclic references. However, they rely on the definition `eq_of_heq`, which, as defined in the Lean standard library, requires K-like reduction in order to type (this relates to the UIP requirement on the target intensional theory described by Winterhalter et. al. [20]). In translating from Lean to Lean$^-$, implicit uses of K-like reduction are made explicit. and so the use of UIP in Lean$^-$'s definition of `eq_of_heq` must also be made explicit.

Therefore, translating Lean's definition of `eq_of_heq` to Lean$^-$ has the risk of introducing cyclic references. In particular, the translation as currently implemented always uses `castHEq` – whose definition references `eq_of_heq`– to explicitly align types, even when it is not strictly necessary to use heterogeneous equality in the first place.This creates a dependency cycle (`eq_of_heq` → `castHEq` → `eq_of_heq`). To get around this issue, for now we have chosen to manually translate the lemma, making use of the extra lemmas `appArgHeq` and `forallEqUV'` – `Eq`-adapted forms of the corresponding `HEq` congruence lemmas – avoiding the use heterogeneous equality in the proof. This leaves us with the following three "bootstrapping lemmas":

```
theorem appArgEq {A : Sort u} {U : Sort v}
  (f : (a : A) → U) {a b : A} (hab : a = b) : f a = f b := ...
theorem forallEqUV' {A : Sort u} {U V : A → Sort v}
  (hUV : (a : A) → U a = V a) : ((a : A) → U a) = ((b : A) → V b) := ...
-- manual translation of stdlib's definition of `eq_of_heq` to Lean-
theorem eq_of_heq {A : Sort u} {a b : A} (h : a == b) : a = b := ...
```

The translation then overrides the standard library's definition of `eq_of_heq` with this one, as opposed to creating a fresh translation-specific definition. This is done as a convenience for the congruence proofs, whose tactics (e.g. the `subst` tactic) produce uses of `eq_of_heq`.

D Producing Equality Proofs

With respect to the modified functions checking for definitional equality between terms, most of them combine and propagate equality proofs that are produced by their subroutines and do not produce proofs themselves at a "base level". The three functions that do generate such "base proof terms" are `isDefEqProofIrrel`,

[19] The proofs themselves can be found here: `https://github.com/rish987/Lean4Less/blob/main/patch/PatchTheorems.lean`

`toCtorWhenK`, and `isDefEqFVar`; our modifications to these functions are described below.

We adapt the kernel function `isDefEqProofIrrel`, which checks whether two proof terms are equal by proof irrelevance (if they have Lean-defeq propositional types), to generate a proof of equality between the proof terms using the `prfIrrel` axiom. If `isDefEq` returns a proof of equality between the propositional types, this means that proof irrelevance/K-like reduction was used at some point in the equality check, so these propositions might[20] not be definitionally equal in Lean$^-$. In such a case, therefore, we would need to we use the proof irrelevance lemma `prfIrrelHEqPQ` which takes this explicit proof of equality between the LHS and RHS proof types (and returns a heterogeneous equality proof). Otherwise, we can return a proof using `prfIrrelHEq`, which assumes that the LHS and RHS propositions are the same. As an additional optimization, this function may also return `none` if the proofs themselves are computably Lean$^-$-defeq in a small number of steps.

We generate a similar proof in the case of K-like reduction. The function `toCtorWhenK`, called by recursor reduction function `inductiveReduceRec`, generates a proof of equality between the major premise `e` of a K-like inductive recursor application and the unique constructor application implied by the inferred K-like type of `e`, which is then substituted in for `e` in the term being reduced in order to continue the reduction (as in the proof of `K.KLR` in Section 2.1). This proof is a direct use of proof irrelevance (recall that K-like inductives must live in `Prop`), and, similarly to the proof irrelevance check in `isDefEqProofIrrel`, may use `prfIrrelHEqPQ` if the types of `e` and the unique constructor application are not Lean$^-$-defeq.

Another place where we may generate base equality proof terms is in equating pairs of free variables introduced by the variable-binding proof arguments of certain congruence lemmas: specifically, `hUV` in `forallHEqABUV'` and `appHEqABUV'`, and `hfg` in `lambdaHEqABUV'`. We "register" the variables as being provably equal in the translation's monadic context:

```
structure TypeChecker.Context : Type where
  ...
  -- stores fvar triples as the map (x : A), (y : B) -> (hxy : x == y)
  eqFVars : Std.HashMap (FVarId × FVarId) FVarId := {}
  ...
```

(corresponding to the triple-valued context computed by the "`Pack`" function of Winterhalter et al. [20]), and add a free variable-specific equality check that returns an equality proof using the relevant variable equality hypothesis.

[20] Note that `isDefEq` might produce equality proofs even if the terms are already Lean$^-$-defeq– specifically, this happens when proof irrelevance is used in a "non-essential" way in checking that they are definitionally equal.

E Adding Extensionality to Lean

Given that Lean4Less's implementation is motivated by a general ETT-to-ITT translation, an interesting prospect is the possiblilty of adaptating it for the purpose of translating Lean terms from some more powerful theory that has been extended with additional definitional equalities back to the original theory. Indeed, as Lean4Less is implemented in Lean, it could be modified to make it capable of eliminating general, user-defined equalities beyond those already defined in Lean. That is, it could accept input terms from some hypothetical user-defined extensional theory "Lean_{e^*}", which is Lean extended with some kind of limited equality reflection rule:

$$\frac{\Gamma \vdash_{e^*} A : \texttt{Sort u} \quad \Gamma \vdash_{e^*} t, u : A \quad \text{compeq}(\Gamma, A, t, u)}{\Gamma \vdash_{e^*} t \equiv u} \ [\text{RFL}^*]$$

where the $\text{compeq}(\Gamma, A, t, u)$ criteria states that, in context Γ, t `==` u is provable automatically in Lean, due to it having been registered directly by a user, or being derivable from other registered equalities. The Lean kernel itself could then be extended to accept user-defined extensional equalities, with the assurance that it will be possible to translate it back to Lean's theory via the modified Lean4Less translation. On the other hand, if we wish to continue using the current Lean kernel, another option is to integrate Lean4Less with existing elaboration routines to allow for a real-time translation that would simulate native kernel support for extensional reasoning.

Regarding the user input of extensional equalities, it will be important to distinguish between "directed" and "undirected" equalities. Undirected equalities are analogous to proof irrelevance, unit-η and function-η in Lean, (and are implemented in the Lean4Lean typechecker kernel's `isDefEqCore` function). Suppose we have a hypothetical constant annotation `@[deq]` thats marks an equality theorem as an extensional definitional equality that "known" to the kernel. This would allow us to prove the following theorem by reflection:

```
@[deq]
theorem addComm (x y : Nat) : x + y = y + x := ...
example (x y z : Nat) : x + (y + z) = x + (z + y) := rfl
```

Here, Lean checks the definitional equality of the arguments in turn, invoking [RFL*] via `addComm` on the second argument of the outermost addition. However, an undirected definitional equality would *not* allow us to prove:

```
-- (Lean's addition function matches on the second argument,
-- so this does not hold definitionally)
@[deq]
theorem incEq (x : Nat) : 1 + x = Nat.succ x := ...
-- cannot be proven with `rfl`
example (x y : Nat) : y + (1 + a) = Nat.succ (y + a) := sorry
```

The problem is the following: for the outermost application to reduce, the second argument's weak-head normal form must be an application of `Nat.succ`, which is not the case for `1 + a`. While `1 + a` is definitionally equal to `Nat.succ a` by `incEq`, this equality does not apply when computing its weak-head normal form.

For this, we instead require a "directed equality" (a.k.a. "rewrite rule") that can be applied during reduction, allowing us to "rewrite" the addition to a constructor application. Let us use the hypothetical annotation `@[drw]` to register a directed extensional equality theorem, enabling here a proof by `rfl`:

```
@[drw]
theorem incEq (x : Nat) : 1 + x = Nat.succ x := ...
example (x y : Nat) : y + (1 + a) = Nat.succ (y + a) := rfl
```

Directed equalities may seem to be strictly more powerful than undirected ones, but they are only practically applicable as long as they satisfy the properties of termination and confluence, which are well-studied in other systems such as Dedukti [4] where rewrite rules are built-in. Without a terminating set of rewrite rules, typechecking/elaboration will also not terminate (for instance, it would not be acceptable to register the commutativity of addition as a directed equality). Confluence is an important property in ensuring that the user-provided reduction rules are unambiguous – in particular, it ensures that definitional equality checking via comparison of normal forms effectively decides equational theory that they define[21]. Termination and confluence must also be considered in light of the reduction rules that Lean natively implements, namely those of recursor, K-like, struct-like and quotient reduction.

Regarding Lean's `cc` / `grind` Tactic Some of the functionality suggested above for allowing Lean to decide a larger class of definitional equalities may be reminiscient of automation already present in Lean for congruence closure [15], which was first introduced in Lean 3's `cc` tactic, and more recently superceded by Lean 4's `grind` tactic. Lean's congruence closure procedure uses a powerful algorithm widely used by SMT solvers that attempts to find an equality proof between two specified terms, taking local equality assumptions into consideration. The fact that automation already exists for this purpose may bring up some questions regarding what potential "benefits" an approach for equality proof reconstruction based on an extensional-to-intensional translation may have over existing, more well-established approaches such as congruence closure.

For instance, one could imagine translating from an "extensional" version of Lean in which the elaborator automatically calls a congruence closure algorithm whenever a typing discrepancy is encountered, and, if the algorithm returns a proof, uses the returned proof to "patch up" the discrepancy via a type cast (as is already implemented in Lean4Less) to help build a finally elaborated term. Such

[21] The equational theory defined by a rewrite system is the reflexive, transitive, symmetric closure of the relation between terms defined by the individual rewrite rules.

an approach could work in principle, however from a practical perspective it is hardly reasonable. An implicit tradeoff that many proof assistant kernels have to make is between providing convenient automation that allows the kernel to identify as many equal terms as possible (avoiding the need for users to manually provide equality proofs), and providing timely negative feedback in the event of a typing error. From a user perspective, it would be unacceptable to call the equivalent of Lean's `grind` tactic to try to resolve every single instance of a typing discrepancy that is encountered. These tactics are much better suited for when the user already heavily suspects that equality can be proven beforehand.

The approach we suggest is rather to extend the existing kernel `isDefEq` routine in simple, limited, and efficient ways, allowing it to identify a larger class of provably equal terms while minimally sacrificing the responsiveness of the system in the event of ill-typedness. The proof reconstruction algorithm we could implement for translating from this extensional version of Lean could then simply extend on the implementation we already have for Lean4Less's `isDefEq` function. While this may not cover as much as an approach based on a full-blown congruence closure algorithm in terms of enabling more definitional equalities, it could be a very reasonable comprimise allowing for some level of user-specified definitional equalities while still providing timely negative feedback to the user.

F Meta-Theoretical Implications for Lean

The result by Hofmann showing conservativity of ETT over ITT [8], as well as the work by Winterhalter et. al. [20] were done with respect to particular theories that do not exactly coincide with the theories Lean_e^- and Lean^- that we discussed in this paper. However, we conjecture that these results can be extended to apply to our theories, with the following relative conservativity property of Lean_e^- over Lean^-:

Conjecture 1. For all valid Lean^- typing contexts Γ and Lean^--typeable terms T, if there is some term t such that $\Gamma \vdash_e^- t : T$, then there is some term t' such that $\Gamma \vdash^- t' : T$.

Unfortunately, Lean itself is not technically conservative over Lean^-: proof irrelevance enables us to construct proofs of some propositions that would be impossible to prove otherwise, unless we explicitly assume the presence of the axiom `prfIrrel` in the Lean^- typing context. We conjecture that this assumption is sufficient for showing an adjacent property:

Conjecture 2. For all valid Lean^- typing contexts Γ and Lean^--typeable terms T, if there is some term t such that $\Gamma \vdash t : T$, then there is some term t' such that $\texttt{prfIrrel} :: \Gamma \vdash^- t' : T$.

In light of this, we may wonder about the extent to which Lean^- can be used as a "proxy metatheory" for Lean for the purpose of simplifying meta-theoretical analyses. Carneiro's Lean4Lean project to formalize Lean's metatheory [6] faces

some difficulties in consistency analyses particularly attributable to features such as definitional proof irrelevance and K-like reduction, and the more recent features of "struct eta" and "struct-like reduction" (whose elimination should also be under the scope of the Lean4Less translation). So, certain important meta-properties of Lean may be significantly easier to show in the smaller theory of Lean$^-$ where these problematic features have been removed[22].

Conjecture 2 is particularly interesting with respect to the consistency property, since it would imply that any axiom-free proof of the proposition `False` in Lean would translate into a proof of `False` in Lean$^-$. Therefore, the consistency of Lean$^-$– assuming that the axiom `prfIrrel` is in the typing context – would imply the consistency of Lean. Proving Conjecture 2 formally in Lean could be done constructively via a proof of correctness of the translation implemented by Lean4Less. If we can prove in Lean that Lean4Less's implementation ensures the property that on all well-typed, terminating Lean input environments, the translation terminates and produces well-typed, semantically equivalent Lean$^-$ output environments, could use this property to formally prove the equivalent of Conjecture 2 in Lean.

Attempting such a proof of program correctness could also be quite interesting in relation to identifying potential consistency bugs in Lean's kernel implementation. The ability to eliminate a certain definitional equality from Lean testifies the fact that its inclusion as a convenience for formalization does not expand the class of provable propositions in any meaningful way, so in particular there is no possibility of it introducing consistency issues. If we encounter any significant difficulties in trying to construct this proof, it may in fact be the case that Conjecture 2 is not provable w.r.t. the type theory that is actually implemented by the Lean kernel, pointing to a possible issue in the kernel implementation of one of the eliminated definitional equalities that renders Lean inconsistent.

Another relevant and interesting task is to formalize the correctness of the Lean kernel relative to Lean's type theory. In combination with the proof of translation correctness w.r.t. the Lean and Lean$^-$ type theories, doing so would formally justify the continued use of the Lean kernel while reasoning about the meta-theoretical results at the level of the Lean$^-$ theory. Assuming we are able to show that Lean$^-$ is consistent[23], Lean$^-$ becomes an ideal target for us to translate Lean formalizations to, in order to typecheck terms with this smaller,

[22] In particular, previous examples of non-termination and undecidability of typechecking shown by Carneiro [6,5] have depended on the use of definitional proof irrelevance and K-like-reduction. These features do not exist in Lean$^-$, so it is an open question whether or not the same issues affect the smaller theory of Lean$^-$. We conjecture that both decidablilty of typechecking and termination may in fact hold – if still not entirely, then perhaps at least with much weaker assumptions – without K- and struct-like reduction.

[23] Note that this cannot be done in Lean itself, as this would violate Gödel's Incompleteness Theorem, so such a result would either have to be shown informally or in a different system.

provably safe kernel. In this respect, it will also be interesting to formally prove that the Lean$^-$ kernel implementation is correct according to the Lean$^-$ type theory.

To have this extra degree of assurance for large-scale formalizations such as Mathlib, it may at first seem necessary to hyper-optimize the Lean4Less translation so that it can effectively scale up to the translation of Mathlib-size formal libraries. However, we probably do not need to actually go this far. It is important for our translation to be practical to some extent, but really only insofar as it is capable of translating a select few proofs to Lean$^-$: specifically, the proofs of correctness of the Lean kernel, Lean4Less translation, and Lean$^-$ kernel implementations. Having done so, we can be almost just as confident in using the original Lean kernel to typecheck large formalizations as we would be if we were to translate the entire libraries themselves to be typechecked with the Lean$^-$ kernel. This would not afford *exactly* the same degree of confidence, as there would still be some additional trust that we would implicitly place on the stack of tooling used to generate executable machine code from Lean code (in particular, the Lean code generator and the C compiler) in preserving the operational semantics of the programs that we originally implemented and reasoned about in Lean.

Additionally, we can think about *extending* Lean's type theory and typechecker kernel in certain conservative ways (perhaps along the lines of what was described in Appendix E) to obtain some hypothetical theory "Lean$^+$", and formally proving the correctness of a Lean$^+$ typechecker kernel and a translation from Lean$^+$ to Lean. Such a translation could then be composed with the translation from Lean to Lean$^-$ for a provably correct translation from Lean$^+$ to Lean$^-$, with these proofs (possibly first expressed in Lean$^+$) translated to Lean$^-$ via this composite translation, giving us the confidence to reason henceforth with a typechecker kernel that decides this more powerful theory (again, at the cost of some extra trust in the stack of tooling that generates the machine code from the formal representation of the implementation in Lean$^+$).