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

Rishikesh Vaishnav

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

Abstract. Lean is a proof assistant whose type-theoretic foundations are based on the calculus of inductive constructions, one of its most prominent features being its small typechecking kernel that aims to be as minimal as possible while still being convenient enough to use for higher-level formalizations. It takes after the proof assistant Coq in many aspects of its theory and design, but notably differs in its use of a number of additional definitional equalities, particularly those of proof irrelevance and "K-like reduction", which are great conveniences for mathematical formalization but complicate meta-theoretical analyses and the task of exporting proofs from Lean to other proof assistants.
In this paper, we describe a translation of Lean proofs to a smaller theory "Lean$^-$", with fewer such definitional equalities, following a translation from extensional type theory to intensional type theory where the implicit use of definitional equality in type conversion is replaced by the explicit use of a type cast. The translation has been implemented in Lean itself in a tool called Lean4Less, which is able to successfully translate certain libraries (e.g. the Lean standard library) to the Lean$^-$ theory. 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 [14] is a proof assistant developed by the Lean FRO whose type-theoretic foundations are based on the Calculus of Inductive Constructions [16] and share many similarities with that of the proof assistant Rocq [7]. It has become especially popular with mathematicians in recent years, being well-known for its "Mathlib" library [9,8], a large and quickly growing body of mathematics formalized in Lean. It features a small, fast kernel that attempts to be a "minimal" foundation for sound typechecking of Lean proofs. Given Lean's popularity, it is of high interest to export Lean proofs to other proof assistants in order to both allow for more confidence in their correctness by typechecking them with a separate kernel, and provide other proof assistant communities with access to Lean formalizations.

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 (sometimes abbreviated in this paper as "defeqs"), 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. We note that the translation task this entails can be interpreted as a special case of a translation from "Lean^-_e", which is Lean$^-$ extended with an extensional reflection rule. In Section 2, we provide an overview of how we have implemented our tool as a modification of Lean4Lean, an external typechecker for Lean implemented in Lean, and how we verify our translation output. In Section 3, we provide more details on the implementation of the translation. In Section 4 we describe some translation results on specific libraries, providing data regarding translation overhead and runtime. Finally, in Section 5, we discuss 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[1]:

$$\frac{\Gamma \vdash P : \texttt{Prop}^2 \quad \Gamma \vdash h : P \quad \Gamma \vdash h' : P}{\Gamma \vdash h \equiv h'} \ (\mathrm{PI})$$

[1] The full set of typing rules in Lean can be found in [5].

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

Proof irrelevance is useful, for example, in establishing the definitional equality of subtype instances with definitionally equal values, but differing membership proofs. Subtypes in Lean are defined with the inductive type:

```
-- subtype inductive type
-- (curly brackets `{...}` denote auto-inferred arguments)
inductive Subtype {A : Sort u} (p : A → Prop) where
| mk : (val : A) -> (property : p val) : Subtype p
```

Lean's equality inductive type `Eq` has the constructor:

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

which requires definitionally equal LHS and RHS in its output type (the `#check` command checks that a term has a specified inferred type, here we use it to display the type of `Eq.refl`). Suppose that we define the following subtype for natural numbers less than 5, along with a (pseudo-)constructor:

```
def NatLT5 : Type := Subtype (fun n => n < 5)
def NatLT5.mk (n : Nat) (p : n < 5) : NatLT5 :=
  @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[3]. However, optional support for definitional proof irrelevance has recently been added with the `SProp` type [4]. Agda supports user-annotated irrelevant function arguments and struct fields[5], while F* erases the details of SMT solver-generated equality proofs [19]. The PVS proof assistant[6] features a special case of proof irrelevance in its handling of the equality of predicate subtype constructions.

1.2 K-Like Reduction

Lean also features a reduction rule called "K-like reduction" (KLR), enabling recursor reduction to proceed as it would under axiom K for equality types, a.k.a.

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

[3] https://rocq-prover.org/doc/V9.0.0/stdlib/Stdlib.Logic.ProofIrrelevance.html

[4] See `https://rocq-prover.org/doc/V9.0.0/refman/addendum/sprop.html`.

[5] See `https://agda.readthedocs.io/en/v2.5.4/language/irrelevance.html`.

[6] `https://pvs.csl.sri.com/`

uniqueness of identity proofs (UIP). However, Lean enables K-like-reduction for *all* so-called “K-like” inductive types with a single constructor that has no non-index arguments. For example, this K-like inductive definitionally satisfies axiom K thanks to proof irrelevance:

```
inductive T : Prop where | mk : T
theorem T.K (t : T) : t = T.mk := rfl
```

The reduction rule for `T` gives us the definitional equality:

```
#check (T.rec : {m : T → Sort u} → m T.mk → (t : T) → m t)
example : T.rec true T.mk = true := rfl
```

Normally, such reductions are limited to explicit constructions in the major premise argument (`T.mk` above). However, thanks to K-like reduction, we also have 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, without needing an explicit construction – here, the major premise argument is simply the variable `t`. While reducing the recursor application, the kernel is able to “rewrite” `t` to `T.mk` during reduction (an operation that is justified by proof irrelevance), allowing the LHS to reduce to `true`.

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[7].

1.3 Meta-Theoretic Challenges

While proof irrelevance and K-like reduction are crucial to enabling convenient mathematical formalization in Lean, 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. As described by Carneiro [6], these features, along with the related features of struct eta and struct-like reduction, produce significant difficulties in our ability to formalize consistency results regarding Lean’s meta-theory.

Such definitional equalities also complicate the task of exporting Lean proofs to other proof assistants, which is an important task 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

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

that of Gilbert in translating Lean to Rocq,[8] 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 [13], an encoding of the general case of proof irrelevance within Dedukti may not be possible.

Considering the above difficulties, one may wonder whether or not it is possible to "eliminate" certain 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 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.

1.4 Target Theory: Lean^{-}

To this end, we propose our target theory Lean^{-}, where definitional proof irrelevance has been replaced with an axiom:

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

Using this theorem, we can also eliminate K-like reduction, which becomes a provable proposition in this smaller theory:

```
inductive T : Prop where | mk : T -- K-like inductive type
#check (T.rec : {m : T → Sort u} → m T.mk → (t : T) → m t)
theorem T.KLR (t : T)
  : T.rec true t = 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 user-provided type annotation or the typing constraints of the surrounding term. For

[8] https://github.com/SkySkimmer/rocq/tree/lean-import

example, we can eliminate proof irrelevance when it is used directly, as in the following example:

```
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 transports 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
```

We can also eliminate K-like reduction using `prfIrrel`, as shown above.

Theoretical Presentation In our target theory Lean^{-}, we have removed PI and KLR from Lean's type theory. We would like to define some translation $|\cdot|$ on Lean-typeable terms that satisfies a certain soundness criterion:

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

where $\Gamma \vdash^{-} t : T$ is the notation for Lean^{-}'s typing judgment, and $|\Gamma|$ simply applies the translation to every type in the context Γ. In words, we want the translation of a term $t : A$ in Lean to produce some term that is well-typed in Lean^{-} as the translation of A.

However, this property alone is not sufficient for our purposes. We would also like to ensure that some notion of type semantics is 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 the property that all translated term are the same as the originals except possibly also being "decorated" with type casts. We capture this notion with the similarity relation "$\sim$" defined in [20]. Specifically, we want our translation to satisfy the property that, for all Lean-typeable t, we have $t \sim |t|$. Such a property also allows translated terms to easily be translated back to the original theory by simply removing the casts.

1.5 A Middle-Ground Extensional Theory: Lean$^{-}_{e}$

The above suggests that a translation from Lean to Lean^{-} may be feasible through the use of type casting, but the question remains whether this can be done in general. 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)
```

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 the proof irrelevance example 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* propositional equality to be considered definitional[9].

To obtain this more general extensional theory to translate from, we can add the equality reflection rule below to Lean$^-$, obtaining the extensional theory "Lean$_e^-$":

$$\frac{\Gamma \vdash_e^- A : \mathsf{U}_\ell \qquad \Gamma \vdash_e^- t, s : A \qquad \Gamma \vdash_e^- \_ : t =_A s}{\Gamma \vdash_e^- t \equiv s} \tag{RFL}$$

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:

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

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

So, because we have for any t, $\Gamma \vdash t : A \implies \Gamma \vdash_e^- t : A$, we can reformulate our problem as one of finding some translation $|\cdot|$ such that:

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

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

[9] 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* [19], and Nuprl [2] restrict RFL to some subset of provable propositional equalities.

work by Oury [15] and Hofmann [12], with the first result showing conservativity of ETT over ITT demonstrated by Hofmann [11] (we conjecture that this result can be adapted to show conservativity of Lean over Lean$^-$).

This translation places certain restrictions on the target intensional theory, namely that it exhibits propositional UIP and function extensionality. Our target theory of Lean$^-$ satisfies UIP thanks to the axiom `prfIrrel`, of which UIP is a special case:

```
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 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 3.1).

Our theories can be summarized in the following table:

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

Practically speaking, our translation does not need to handle the full ETT-to-ITT translation – because we only care about translating Lean-typeable terms, all we have to consider is the singular case of proof irrelevance, interpreted as a particular application of extensional reflection. 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 (an example of this is shown in Appendix A).

Such a translation must generalize from the equality type `Eq` to the heterogeneous equality type `HEq`, which differs in being 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
```

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 proofs of translation correctness, it is less convenient for an actual implementation (where it is not necessary to prove equality of these types), so we instead choose to return to the "John Major equality" used by Oury [15], which is a more compact and equivalent formulation already defined in the Lean standard library in the `HEq` type above (`JMeq` in the Rocq standard library).

2 Implementation Overview

2.1 Adapting `ett-to-itt`?

Although a Coq-verified translation from from ETT to ITT already exists in the `ett-to-itt` repository [21], which could be extracted to an executable OCaml program [10] and possibly used in our translation, there would be a number of challenges associated with this approach (described in more detail in Appendix B). In particular, the extracted code would require as input some representation of Lean typing derivations, which Lean currently provides no way to obtain. Therefore, 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 3.3). 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 3.2) – effecting, in this way, a kind of "patching" typechecker.

2.2 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 MetaCoq project [18]. 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. An implementation in Lean also 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 primarily consisting of the following three 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. It is a subroutine of `isDefEq`, where terms must sometimes be (partly) reduced to determine if they are defeq.

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 `HEq t s`
def whnf (e : PExpr) : RecM (PExpr × Option EExpr) := ...
                                 -- ^ proof of `HEq e (whnf e)`
```

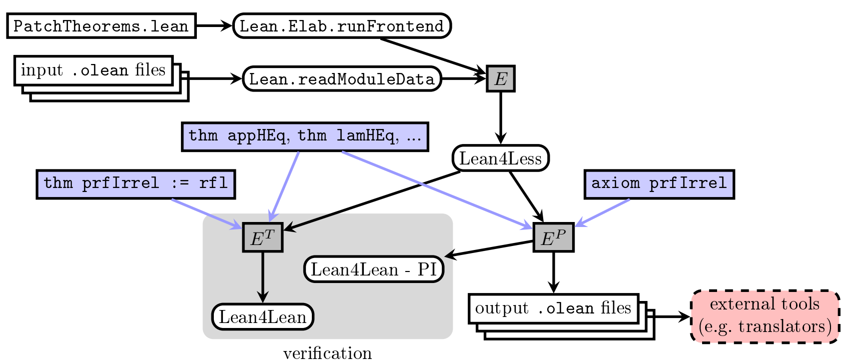
(`PExpr` and `EExpr` are Lean4Less-specific types for representing translated terms and equality proof, respectively). All three functions *optionally* return a patched expression/equality proof depending on whether proof irrelevance was used in typing/defeq-checking. If it was not, then we return `Option.none`, indicating that no equality proof/translation was required – in the case of `inferType`, this means that `e` is already well-typed in Lean$^-$, and in the case of `isDefEq`, that means that `t` and `s` already definitionally equivalent in Lean$^-$.

The `inferType` function now may also return a translated version of the input expression injected with transports where required by typing constraints (see Section 3.2) – 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 the definitional equality between the applications `f a` and `f b`, 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 (the `HEq` equivalent of) Lean's `congrArg` lemma in order to produce the proof term `congrArg f a b p : f a = f b`.

2.3 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 1.4, the output of our translation should only "decorate" the input with casts between types that are already Lean-defeq. To verify this property, we generate a verification environment E^T containing equality theorems between the original and translated types of every defined constant, which are proven by reflection. We then typecheck E^T with the normal Lean kernel. Our translation and verification workflow is summarized in the diagram below:



3 Implementation Details

3.1 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[10]. In particular, we need a set of "congruence lemmas" to compose equality proofs

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

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 : HEq A B) (hUV : (a : A) → (b : B) → HEq a b → HEq (U a) (V b))
  : HEq ((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 : HEq A B) (hfg : (a : A) → (b : B) → HEq a b → HEq (f a) (g b))
  : HEq (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 : 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) := ...
```

`appHEqABUV'` contains the additional hypothesis `hUV` that allows us to equate `U` and `V` in the 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[11]. 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:

```
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[12].

In addition, we also need the proof irrelevance axiom and its extension to heterogeneous equality with equal proof types, and, for convenience, we add a heterogeneous cast function:

```
axiom prfIrrel {P : Prop} (p q : P) : Eq p q
theorem prfIrrelHEq {P : Prop} (p q : P) : HEq p q := ...
```

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

[12] If we assume this axiom in our theory, 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`.

```
theorem prfIrrelHEqPQ {P Q : Prop} (hPQ : HEq P Q)
   (p : P) (q : Q) : HEq p q := ...
def castHEq {A B : Sort u} (h : HEq 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).

3.2 Producing Patched Terms

During translation, the output is obtained by "injecting" type casts into the terms in places where expected and inferred types are not Lean^{-}-defeq. Expected type requirements can arise either from user-provided annotations or from typing restrictions imposed by Lean's type theory. These type casts require a proof of equality between their expected and inferred types, which is computed with a call to `isDefEq`. More details on this computation are provided in Appendix D.

User-provided type annotations can come from constant signatures or let bindings. 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 environment. Checking let bindings, on the other hand, occurs as a subroutine of type inference.

Type casts may also be inserted on account of the following typing rules used by Lean:

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

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

The rules `LAM`, `PI`, and `LET` require the binder types (and output type, in the case of `PI`) 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 the `APP` case above, where the domain type of the function must definitionally match the inferred type of the argument, and the argument may be cast if this is not the case in Lean^{-}. 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 `L4L.castHEq` cast function.

3.3 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. 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 implements a "lazy δ-reduction" optimization in its `isDefEq` check in which it avoids expanding 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 producing a larger proof resulting from a more detailed comparison of subterms.
- Lean's kernel makes use of a cache for recording previously computed weak-head normal forms. Lean4Less adapts this cache to store a reduction proof in addition to the weak-head normal form itself, and can be queried before attempting a WHNF computation to avoid an unnecessary computation. This translates into an output optimization since these redundant proofs will also share object pointers in the output.

Lean4Less implements a number of additional optimizations of its own (not described here).

4 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 2.3. 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)[13]
`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. 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. This may also be addressed through more efficient use of caching, for example in `EExpr.toExpr` computations, which we have observed to take up a disproportionate amount of computation time.

5 Prospects

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.

Another potential benefit of having a translation from Lean to Lean$^-$ is that it can greatly simplify meta-theoretical analyses of Lean's type theory by enabling us to use Lean$^-$ as a "proxy theory" for Lean itself. The general idea is that any meta-theoretical result shown for Lean$^-$ would automatically transfer to Lean, provided the correctness of the translation implementation. More details are provided in Appendix F.

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

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.

6 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 that has been implemented in Lean. Our tool, "Lean4Less", has been successfully able to translate certain medium-sized libraries, and we hope to soon scale up our translation to handle larger formalizations. We believe that this work sets the foundation for the first practical implementation of a general 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 and typecheck them with the same small, trusted kernel. Additionally, while this work primarily concerns a particular implementation of a extensional-to-intensional translation applied specifically to eliminating the use of proof irrelevance in the typing of Lean terms, the techniques and optimizations developed here could be transferrable to similar translations implemented for other proof assistants, either for the purpose of proof export/translation or for extending them to have extensional-like features of their own.

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 (2018)

4. Blanqui, F., Dowek, G., Grienenberger, É., Hondet, G., Thiré, F.: A modular construction of type theories. CoRR **abs/2111.00543** (2021)
5. Carneiro, M.: The Type Theory of Lean. Master's thesis (2019)
6. Carneiro, M.: Lean4lean: Towards a formalized metatheory for the lean theorem prover (2024)
7. Community, C.: coq
8. community, M.: mathlib4 (Github)
9. mathlib community, T.: The Lean mathematical library. CoRR **abs/1910.09336** (2019)
10. Forster, Y., Sozeau, M., Tabareau, N.: Verified Extraction from Coq to OCaml (Jun 2024). `https://doi.org/10.1145/3656379`
11. 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)
12. Hofmann, M., Rijsbergen, C.J.: Extensional Constructs in Intensional Type Theory. Springer-Verlag, Berlin, Heidelberg (1997)
13. Hondet, G., Blanqui, F.: Encoding of Predicate Subtyping with Proof Irrelevance in the $\lambda\Pi$-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`, `https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.TYPES.2020.6`
14. Moura, L.d., Ullrich, S.: The lean 4 theorem prover and programming language. In: Automated Deduction – CADE 28. pp. 625–635 (2021)
15. 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`
16. 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)
17. Selsam, D., de Moura, L.: Congruence closure in intensional type theory. CoRR **abs/1701.04391** (2017)
18. 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`
19. 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`
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)
21. Winterhalter, T., Tabareau, N.: ett-to-itt (Github)
22. Winterhalter, T., Tabareau, N.: ett-to-wtt (Github)

A A Complex Lean⁻ 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) → 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' := ...
-- proved using `prfIrrel`
theorem prfIrrelHEqPQ {P Q : Prop} (h : P = Q) (p : P) (q : Q) : HEq p q := ...

variable (P : Prop) (Q : P → Prop) (p q : P) (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
  (appHEq (congrArg Q (eq_of_heq (prfIrrel p q)))
    (fun _ _ _ => HEq.rfl)
    (appHEq rfl ... HEq.rfl (prfIrrel rfl p q))
    (prfIrrelHEqPQ (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`. The initial proof of `T p = T q` is straightforward, but at this point the remaining arguments `Qp : Q p` aand `Qq : Q q` have non-Lean⁻-defeq types. Therefore, we must use the more general lemma `prfIrrelHEqPQ` that requires a proof of equality between `Q p` and `Q q`. Correspondingly, as the domain types of the function are not Lean⁻-defeq, we must generalize our application congruence lemma to `appHEq`.

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

- The translation formalized by `ett-to-itt` takes as input extensional typing derivations, rather than extensional terms. Lean currently has no output or representation of typing derivations, so we would have to construct these derivations ourselves, likely through the modification of a kernel implementation – however, as described below, 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 representation.

– `ett-to-itt`'s input derivations are assumed to come from a *minimal extensional theory*, which is not a superset of Lean's theory. To align a Lean derivation with the expected input theory, we would have to do some preliminary alignment on the Lean 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 + the axiom `prfIrrel`.
– `ett-to-itt`'s output takes the form of terms typeable in a *minimal intensional theory*. We would correspondingly have to do some post-processing on this output in order to recover the typeability of the terms without extra axioms/lemmas in the target theory, "undoing" our work preprocessing the derivations (where we originally introduced uses of these extra axioms/lemmas).
– 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 3.1 all proven in Lean with the usual high-level Lean tactics[14]. As elaborated, they are in fact already valid Lean$^-$ proofs, 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]; recall from Section 1.5 that UIP holds definitionally in Lean as a special case of proof irrelevance, which is also expressed through K-like reduction. In going from Lean to Lean$^-$, UIP is transformed from a definitional equality into a propositional one, and so the use of UIP in Lean$^-$'s definition of `eq_of_heq` must be made explicit.

Unfortunately, it is not sufficient to simply translate Lean's definition of `eq_of_heq` to Lean$^-$, because this may create cyclic references. In particular, the translation as currently implemented always uses `castHEq` – which references `eq_of_heq`– to apply type transport, even when it is not strictly necessary to use heterogeneous equality in the first place.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

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

HEq congruence lemmas). 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 : Eq a b) : Eq (f a) (f b) := ...
theorem forallEqUV' {A : Sort u} {U V : A → Sort v}
  (hUV : (a : A) → Eq (U a) (V a)) : Eq ((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 : HEq a b) : @Eq A a b := ...
```

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

Note that it is in fact possible to prove every lemma without any definitional equalities whatsoever, relying on syntactic equality alone in the style of "weak type theory" (WTT) – the translation from [20] was extended to translate from ETT to WTT in [22]. As we attempt to eliminate more definitional equalities, it would be convenient if the translation framework could "bootstrap" itself so we do not have to manually eliminate the uses of these definitional equalities. For instance, we could further optimize the output to avoid `HEq` and use `Eq` wherever possible, allowing us to use `cast` instead of `castHEq` (which would be sufficient to automate the translation of `eq_of_heq` in this case). But it is not clear that we can optimize the output to the extent that these cyclic references can be generally avoided altogether as we attempt to eliminate further definitional equalities.

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`, `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, we use the heterogeneous proof irrelevance lemma `prfIrrelHEqPQ`, otherwise we return a proof using `prfIrrelHEq` (when the propositional types are already Lean$^-$-defeq). As an optimization, this function may also return `none` if the proofs themselves are computably Lean$^-$-defeq under a limited recursion depth.

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` to continue the reduction (as in the proof of `K.KLR` in Section 1.4). 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 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 monadic context:

```
structure TypeChecker.Context : Type where
  ...
  -- stores fvar triples as the map (x : A), (y : B) -> (hxy : HEq 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 an fvar-specific equality check "`isDefEqFVar`" that returns an equality proof using the relevant variable equality hypothesis (possibly reversed via `HEq.symm`).

E Adding Extensionality to Lean

Given Lean4Less's implementation of a translation framework based on an general ETT-to-ITT translation, an interesting prospect is the possible adaptation of Lean4Less 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, after some work to make it capable of accepting general, user-defined equalities, accept input terms from some hypothetical user-defined extensional theory "Lean$_{\mathrm{e}^*}$" – that is, Lean extended with some kind of rule for "algorithmic reflection":

$$\frac{\Gamma \Vdash_{\mathrm{e}^*} A : \mathsf{U}_\ell \quad \Gamma \Vdash_{\mathrm{e}^*} t, u : A \quad \mathrm{compeq}(\Gamma, A, t, u)}{\Gamma \Vdash_{\mathrm{e}^*} t \equiv u} \ (\mathrm{RFL}^*)$$

where the compeq(Γ, A, t, u) criteria states that, in context Γ, $t =_A u$ is provable automatically in Lean, due to it having been registered directly by a user, or by being automatically derivable from other registered equalities. The Lean kernel itself could then be extended with extensional reasoning, with the assurance that it will be possible to translate it back to the original intensional theory via Lean4Less. 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 simulates native kernel support for extensional reasoning. See Appendix G for a comparison of this possible approach with existing automation for generating equality proofs in Lean.

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, struct and function eta in Lean. These checks are implemented in the kernel's `isDefEqCore` function, and are performed on terms that are already in weak-head normal form (with the exception of proof irrelevance, which can be checked earlier as its equivalence criterion is based solely on typing). For instance, suppose we have the hypothetical constant annotation `@[deq]` marking an equality theorem as one that "known" to the extensional 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, undirected equality would *not* allow us to prove the following:

```
-- 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
```

Here, we instead require a "directed equality" (a.k.a. "rewrite rule") allowing us to "rewrite" the addition to a constructor application. Let us use the hypothetical annotation `@[drw]` to register a directed equality theorem, enabling here a proof by reflection:

```
@[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 primitive notions. Without termination, elaboration will also not terminate (for instance, it would not be acceptable to register the commutativity of addition as a directed equality).

Termination must also be considered in light of the "rewrite rules" that Lean natively implements, namely those of recursor, K-like, struct-like and quotient reduction. For instance, K-like reduction, when interpreted as a directed equality that is added to Lean^{-}, results in non-termination in Lean when it is combined with the recursor reduction rule for equality, as demonstrated by Carneiro [6] (a result adapted by Abel and Coquand [1]).

F Simplifying Theoretical Analyses

Our ability to use Lean$^-$ as a "proxy theory" for Lean is crucially dependent on the correctness of the translation. Namely, we must show that Lean4Less's implementation ensures the property that on all well-typed, terminating Lean input environments, it terminates and produces well-typed, semantically equivalent Lean$^-$ output environments. Until such a formalization is done, however, our confidence will have to rest on the empirical success of Lean4Less in translating large libraries (such as Mathlib).

In particular, with a verified translation we would have the property that any axiom-free proof of the proposition `False` in Lean would translate into a semantically equivalent and typeable proof of `False` in Lean$^-$. Therefore, the consistency of Lean$^-$ would imply the consistency of Lean. In [6], Carneiro describes some difficulties in soundness analyses raised in Lean in particular due to features such as 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 consistency may be easier to prove in the smaller theory of Lean$^-$ where these problematic features have been removed. 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.

Moreover, having a translation restricting Lean to a smaller subset of definitional equalities could ease the formal verification of an implementation of a smaller canonical typechecking kernel for Lean[15], even before the translation has been formally verified. If one is willing to delegate the Lean4Less translation and the current typechecking kernel to the "untrusted" portion of Lean's code base, the Lean$^-$ kernel can take the place of the current Lean kernel as the "official" trusted Lean kernel, provided that we can show the conservativity of Lean over Lean$^-$ and that the translation is successful in translating any input environment that is checkable with the original kernel.

G Regarding Lean's `cc`/`grind` Tactic

Some of the functionality suggested in Appendix E for allowing Lean to decide a larger class of equalities may be reminiscient of automation already present in

[15] Note that we likely still cannot have a provably terminating typechecker for Lean$^-$. If Lean really is conservative over Lean$^-$ (which would follow from having a verified translation), the termination of a Lean$^-$ kernel would not be possible to prove within Lean itself as this approaches Gödel's Incompleteness Theorem. So unfortunately, it is likely that we would still be limited to reasoning about the Lean$^-$ kernel implementation in terms of partial, rather than total correctness.

Lean for congruence closure [17], first introduced in Lean 3 with the `cc` tactic, and 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 proof of equality 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.

Specifically, one could imagine translating from an "extensional" version of Lean in which the elaborator kernel 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 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 to the user 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 approach may not cover as much in terms of deciding equality as an approach based on a full-blown congruence closure algorithm, it could be a very reasonable comprimise allowing for some level of user-specified definitional equalities while still providing adequately snappy negative feedback to the user.