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

Rishikesh Vaishnav

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

Rishikesh Vaishnav
Université Paris-Saclay, ENS Paris-Saclay, LMF, CNRS, Inria, France
rishikesh-hirendu.vaishnav@inria.fr

1 Introduction

Lean [7] is a proof assistant that is especially popular among mathematicians, with one of the fastest growing formalized mathematics libraries [6]. The latest version of Lean, Lean 4, also provides extensive utilities for metaprogramming and (with a few extensions) is usable as a general programming language, with most of Lean 4's code base being implemented in Lean. Lean's type theory derives from the calculus of constructions with inductive types (CIC) and closely resembles that of Coq [4] (another proof assistant based on CIC), some differences being its non-cumulative universe hierarchy and certain definitional equalities (e.g. proof irrelevance).

We use the notation $\Gamma \Vdash t : A$ for Lean's typing judgment ("t types as A in context Γ"), and the notation $\Gamma \Vdash t \Leftrightarrow u$ for Lean's definitional equality judgment, ("t is definitionally equal to u in context Γ"). From Lean's typing/definitional equality rules, the following two are of particular importance to us:

$$\frac{\Gamma \Vdash t : A \qquad \Gamma \Vdash A, B : \mathsf{U}_\ell \qquad \Gamma \Vdash B \Leftrightarrow A}{\Gamma \Vdash t : B}\ (\text{CONV}) \qquad \frac{\Gamma \Vdash P : \mathsf{U}_0 \qquad \Gamma \Vdash p, q : P}{\Gamma \Vdash p \Leftrightarrow q,}\ (\text{PI})$$

The first is the conversion rule, which allows a term to be typed up to definitional equality. The second is the proof irrelevance rule, which states that any two proofs of the same proposition are definitionally equal (that is, that proofs are "irrelevant" for the purpose of typing).

The complete set of these rules can be found in [2]. These judgments are in fact "algorithmic" variants of the "ideal" judgments $\Gamma \vdash t : A$ and $\Gamma \vdash t \equiv u$ (which are shown in [2] to not be decidable in general). We focus on the algorithmic type system here as it corresponds to what is actually implemented in Lean.

Dedukti

Given the popularity of Lean within the formalized mathematics community, it is of high interest to translate proofs from Lean to other proof assistants in order to make these results available to them and to improve interoperability between these systems. In order to achieve such a translation, it is convenient to first represent the theory of Lean in a logical framework and translate the proofs of Lean into this theory, from which they can be exported to other proof assistants.

Dedukti [1] [8] is one such logical framework, whose type theory is based on the lambda-pi calculus modulo rewriting ($\lambda\Pi/R$). It is designed to ease translation between proof assistants whose type theories can be encoded within Dedukti. Dedukti's typing rules are similar to those of Lean (it has dependent types, though it does not have a universe hierarchy/polymorphism), and its definitional equality is based on β-reduction and a set of user-defined rewrite rules.

Submitted to:
LFMTP 2024

In translating from Lean to Dedukti, any definitional equality that is in Lean but not in $\lambda\Pi/R$ must somehow be encoded with rewrite rules such that a term's β-rewrite normal form is agnostic to this definitional equality. Most of Lean's reduction rules are easy to encode directly, for example those for recursor and projection reduction. However, others may be particularly difficult to encode, for example proof irrelevance (where there is no obvious normal form for proof terms, and a simple proof erasure strategy runs into typing issues) and K-like reduction (a generalization of the axiom K that cannot always be converted into a rewrite rule).

For this reason, rather than translating Lean to Dedukti, we propose to first translate Lean terms to terms that typecheck in the more restricted theory "Lean$^-$", where some definitional equalities have been removed and replaced with axioms typed as corresponding propositional equalities. We use the notation $\Gamma \Vdash^- t : A$ for Lean$^-$'s typing judgment and the notation $\Gamma \Vdash^- t \Leftrightarrow u$ for Lean$^-$'s definitional equality judgment. In our first presentation of Lean$^-$, we will remove Lean's proof irrelevance rule (PI) and replace it with the axiom `prfIrrel`:

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

where $=_P$ represents the type of propositional equality between proofs of proposition P (this axiom is provable in Lean by the reflexivity of equality and proof irrelevance).

2 Connections with Extensional Type Theory

We can define an extensional version of Lean$^-$, which we will call Lean^-_e, and the notation $\Gamma \Vdash^-_e t : A$ and $\Gamma \Vdash^-_e t \Leftrightarrow u$ for its typing and definitional equality judgments. This system additionally includes the rule:

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

known as the "reflection" rule. In Lean^-_e, (RFL) and (PI-) together derive (PI). Therefore, Lean's typing is a strict subset of that of Lean^-_e as every derivation in Lean can be converted to a derivation in Lean^-_e. The differences between these theories are summarized in the table below:

Theory	Rules	Decidable?	$\subsetneq$
Lean$^-$ ($\Vdash^-$)	(PI-)	yes	Lean
Lean ($\Vdash$)	(PI)	yes	Lean^-_e
Lean^-_e ($\Vdash^-_e$)	(PI-), (RFL)	no	

So, translating from Lean^-_e to Lean$^-$ is sufficient to translate from Lean to Lean$^-$. While this may seem at first like a more complex task than translating to Lean$^-$ directly from Lean, the fact that proofs can appear arbitrarily in terms means that (PI) may be used anywhere that (RFL) might be in an extensional theory – so, the fact that we are only erasing the particular case of proof irrelevance does not afford us any simplifications in the translation algorithm.

To achieve this translation, we can look at previous work on the translation from extensional type theory (ETT) to intensional type theory (ITT). An algorithm was described by Winterhalter et. al. in [9]. Additionally, this work was formalized in Coq in `ett-to-itt` [10], so an implementation of this algorithm may be extracted to OCaml, though this will not be immediately usable for us for three reasons:

- It is a function on derivations in a minimal extensional theory – while we can easily translate a Lean derivation to Lean^-_e one, a Lean^-_e derivation may still use additional Lean-specific definitional equalities (for example, eta-equivalence, recursor reductions for inductive types). We will need to additionally patch the Lean^-_e derivation to replace any uses of such definitional equalities with (RFL) applied to a corresponding axiom, and following translation, must replace all uses of these axioms in the output term with reflexivity proofs to obtain a Lean$^-$-typeable term (that uses no axioms other than proof irrelevance).

  This will result in large output terms that use many unnecessary reflexivity proofs, so an important post-processing step will be to reduce these to a minimal form (where transport is only preserved on terms that make use of proof irrelevance in their typing) to obtain our final Lean$^-$ output term.
- Lean does not currently provide a way to obtain objects representing typing derivations. It only provides an environment of fully elaborated constants. One way to obtain the derivations would be to build them alongside typechecking, for example by modifying a Lean typechecker implementation to output derivations from its inference and definitional equality routines.

3 Implementation

While it would be valuable to use a verified tool such as `ett-to-itt` in our translation from Lean to Lean$^-$, the tasks above are not trivial (and result in a less-than-optimal implementation) so for practical reasons we will make our first implementation in Lean exclusively, porting aspects of `ett-to-itt` as necessary. Rather than modifying a typechecker to output derivations for input to `ett-to-itt`, we will build "patched" terms in parallel to typechecking (where the relevant derivation rules will be clear from the implementation), obtaining a "patching typechecker". Another benefit of this approach is that we can ensure that casting and equality proofs are only included where necessary (that is, where Lean$^-$'s typechecking would otherwise fail), giving us a minimal translation (the post-translation optimizations described in [9] only eliminate casts up to α-equivalence of the types being cast).

Lean4Lean

One implementation of Lean's typechecker that is of particular interest to us in this endeavor is Lean4Lean, a port of Lean's C++ typechecker code into Lean 4 [3]. This tool being implemented in Lean gives us easy access to Lean's environment, datatypes, and utilities for constructing expressions, which will be very useful in helping us build the output patched terms/proofs.

Lean4Lean's typechecker implementation (found primarily in the file `Typechecker.lean`) has functions for inference, definitional equality, and weak-head normalization (a subroutine of `isDefEq`):

```
def inferType (e : Expr) : RecM Expr := ...
def isDefEq (t s : Expr) : RecM Bool := ...
def whnf (e : Expr) : RecM Expr := ...
```

We have begun to modify these functions to return an `Option Expr` in addition to their normal return values, corresponding to the patched input term for `inferType`, a proof of equality of the input terms for `isDefEq`, and a proof of equality of the original and weak-head-normalized terms for `whnf`.

Testing

We plan to test our implementation on the mathlib4 library [5], the main repository where results are being formalized in Lean. The input for our translation will be the `.olean` files that are output by Lean,

which represent the environment E of fully elaborated constants. Let K be a function from environments to booleans that represents the typechecking of the Lean kernel. We can assume that $\mathrm{K}(E) = \texttt{true}$.

Let the function P on environments represent our translation, which will run on E to patch its terms and output a set of `.olean` files corresponding to the patched environment $E' = \mathrm{P}(E)$. Note that E' will also include the axiom `prfIrrel` and any congruence lemmas used by the generated equality proofs. Let K^- represent the typechecking of Lean$^-$ (that is, identical to K but without proof irrelevance).

We will then perform the following verification steps on E':

1. Check that E' typechecks in Lean$^-$ (that is, that $\mathrm{K}^-(E') = \texttt{true}$), verifying that a sufficient number of typecasts were inserted to allow the patched terms to typecheck without proof irrelevance, and that the generated type equality proofs used in these casts are well-typed.
2. In the environment $\mathrm{Eq}(E, E')$ which is defined to be equal to E with the addition of `prfIrrel` as a theorem, the congruence lemmas, and reflexive equality proofs between the types of the original constants and their patched versions, check that this is well-typed in the original kernel (that is, that $\mathrm{K}(\mathrm{Eq}(E, E')) = \texttt{true}$). This will verify that the type semantics did not change as a result of patching, which should indeed be the case if the patched terms are simply "decorated" by casts that use proofs that are reflexively true in Lean (that is, if the original and patched terms are related by the similarity relation "~" described in [9]).

Note that these verification steps are only important while developing (to help identify any bugs), however they can later be disabled.

Prospects: pseudo-extensionality

With such a patching framework implemented in Lean4Lean, we can consider extending it to obtain intensional terms from an extensional version of Lean, Lean$_e$, that includes a version of the rule (RFL). While ETT is undecidable in general, it should be possible to write an algorithm for generating a subset of the equality proofs that can be used by (RFL) (so, in effect, the version used by Lean$_e$ will be limited to a set of "computably provable" equalities). If this is successful, it can be optimized and integrated with the existing Lean elaborator and users can be allowed to "register" their own procedures for generating proofs of certain equalities. In this way, users will be able to improve Lean's ability to simulate reasoning as if some equalities are definitional, enabling a kind of "extensible pseudo-extensionality" in Lean.

4 Conclusion

In order to translate from Lean to a logical framework like Dedukti, it is useful to first convert Lean terms into a form that only requires a subset of Lean's definitional equality rules. This work-in-progress report describes an effort to modify an implementation of Lean's typechecker, Lean4Lean [2], to output terms that are well-typed in Lean$^-$, a theory where Lean's proof irrelevance rule has been removed (and which may later be extended to eliminate other definitional equalities). As this translation can be considered a special case of the ETT-to-ITT translation described in [9], the framework established here could also possibly be extended to translate terms from a version of Lean that implements extensional features.

5 Bibliography

References

[1] Frédéric Blanqui, Gilles Dowek, Émilie Grienenberger, Gabriel Hondet & François Thiré (2021): *A modular construction of type theories*. *CoRR* abs/2111.00543. arXiv:2111.00543.

[2] Mario Carneiro (2019): *The Type Theory of Lean*. Master's thesis.

[3] Mario Carneiro (2024): *Lean4Lean: Towards a formalized metatheory for the Lean theorem prover*. arXiv:2403.14064.

[4] Coq Community: *coq*.

[5] Mathlib community: *mathlib4 (Github)*.

[6] The mathlib community (2019): *The Lean mathematical library*. *CoRR* abs/1910.09336. arXiv:1910.09336.

[7] Leonardo de Moura & Sebastian Ullrich (2021): *The Lean 4 Theorem Prover and Programming Language*. In: *Automated Deduction – CADE 28*, pp. 625–635.

[8] R. Saillard (2015): *Type checking in the Lambda-Pi-calculus modulo: theory and practice*. Ph.D. thesis, Mines ParisTech, France.

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

[10] Théo Winterhalter & Nicolas Tabareau: *ett-to-itt (Github)*.