Lean4Lean: Towards a formalized metatheory for the Lean theorem prover

Mario Carneiro
Carnegie Mellon University, Pittsburgh, PA, USA

Abstract

In this paper we present a new "external verifier" for the Lean theorem prover, written in Lean itself. This is the first complete verifier for Lean 4 other than the reference implementation in C++ used by Lean itself, and our new verifier is competitive with the original, running between 20% and 50% slower and usable to verify all of Lean's mathlib library, forming an additional step in Lean's aim to self-host the full elaborator and compiler. Moreover, because the verifier is written in a language which admits formal verification, it is possible to state and prove properties about the kernel itself, and we report on some initial steps taken in this direction to formalize the Lean type theory abstractly and show that the kernel correctly implements this theory, to eliminate the possibility of implementation bugs in the kernel and increase the trustworthiness of proofs conducted in it. This work is still ongoing but we plan to use this project to help justify any future changes to the kernel and type theory and ensure unsoundness does not sneak in through either the abstract theory or implementation bugs.

2012 ACM Subject Classification Mathematics of computing → Mathematical software performance; Software and its engineering → Formal methods; Social and professional topics → Systems analysis and design

Keywords and phrases Lean, proof assistant, external verifier, implementation, metatheory, type theory, proof theory

Supplementary Material The formalization and implementation are available at `https://github.com/digama0/lean4lean/tree/itp2024`.

Funding Work supported by the Hoskinson Center for Formal Mathematics at Carnegie Mellon.

Acknowledgements I would like to thank Jeremy Avigad for their support and encouragement, and Yannick Forster, Matthieu Sozeau and the rest of the MetaCoq team for collaborating and taking interest in this work.

1 Introduction

Lean [7] is a theorem prover based on the Calculus of Inductive Constructions (CIC), quite similar to its older brother Coq [3], but as the name suggests, one of the differentiating aspects was the desire to make the kernel "leaner," relying on simpler primitives while retaining most of the power of the system. This was in particular informed by periodic news of unsoundnesses in Coq[1] due to complications of the many interacting features in the kernel. So from the beginning, Lean advertised its simpler foundations and multiple external verifiers as reasons that it would not suffer from the same troubles. These hopes were broadly upheld: for the entire release history of Lean 3, there were no soundness bugs reported against the kernel (written in C++), and our previous work [4] showed the consistency of Lean with respect to ZFC with n inaccessible cardinals for all $n < \omega$, so the soundness story was fairly strong.

[1] `https://github.com/coq/coq/blob/master/dev/doc/critical-bugs.md`

But times move on, and Lean 4 now exists as an (almost) ground-up rewrite of Lean with most components now written in Lean itself. The kernel was one of the few components that was not rewritten, but it was extended with various features for performance reasons like bignum arithmetic, nested inductive types, and primitive projections, and unfortunately some soundness bugs crept in during the process and Lean's record is no longer spotless. Another thing that was lost during the port was the external verifiers: the whole metaprogramming infrastructure was redesigned so old external verifiers are no longer applicable to Lean 4. While most of the new features can be treated as mere abbreviations, nested inductive types and η for structures significantly impact the theory, and as a result the soundness proof from [4] is no longer directly applicable. (But see [12], which covers some of these modifications.)

To restore the soundness story of Lean to a satisfactory state, we believe it is necessary to bring back the external verifiers. And what better language to do so than Lean itself? There are reasons not to use Lean for a truly *external* verifier (and more external verifiers are coming, in Swift[2] and Rust[3]), but there are also advantages to having a kernel written in Lean:

- The Lean elaborator is already written in Lean as a form of "whitebox automation": users are able to make use of elaborator APIs to write their own tactics with just as much flexibility as the core language itself. Similarly, having a "whitebox kernel" makes it easier for Lean users to query the internal state and understand how Lean reduces or typechecks terms.
- Unlike Rust and Swift, Lean is designed for both programming and proving, which means that a kernel written in Lean can have properties proved about it. And we have good reasons for wanting the kernel to satisfy certain properties…

In this paper, we will present `Lean4Lean`, an external verifier for Lean 4, written in Lean. The verifier itself is more than a prototype: it is fully capable of checking the entirety of `Lean` (the Lean compiler package), `Std` (the standard library), and `Mathlib` (the Lean mathematics library). It is not as fast as the original kernel written in C++ because the Lean compiler still has some ways to go to compete with C++ compilers, but the overhead is still reasonable in exchange for the additional guarantees it can provide, and it is suitable as a complementary step in Lean projects that wish to validate correctness in a variety of ways.

The second part of this project, which is much larger and on which we can only report partial progress, is the specification of Lean's metatheory and verification of the `Lean4Lean` kernel with respect to that theory. This is most similar to the MetaCoq project [11], and amounts to a formalization of the results of [4]. Our principal contributions in this area is a definition of the typing judgment, along with some regularity theorems we can prove and others we can only state, as well as invariants and proofs of correctness for a few of the components of the kernel. The remainder is left as future work.

The paper is organized as follows: Section 2 defines the main typing judgment and describes some of its properties. Section 3 gives the core data structures of the typechecker and how they relate to the typing rules from section 2. Section 4 explains how the global structure of the environment is put together from individual typing judgments. Section 5 explains the complexities associated with supporting inductive types, treated as an extension of the base MLTT theory. Section 6 gives some performance results regarding the complete verifier, and Section 7 compares this project to MetaCoq, a Coq formalization of Coq metatheory and a verified kernel.

[2] `https://github.com/gebner/trepplein`
[3] `https://github.com/ammkrn/nanoda_lib`

2 Base theory

2.1 Expressions

At the heart of the theory is the `VExpr`[4] type, which represents expressions of type theory.

```
inductive VExpr where
  | bvar (deBruijnIndex : Nat)
  | sort (u : VLevel)
  | const (declName : Name) (us : List VLevel)
  | app (fn arg : VExpr)
  | lam (binderType body : VExpr)
  | forallE (binderType body : VExpr)
```

- `bvar` n represents the n'th variable in the context, counting from the inside out. As we will see, Lean itself uses "locally nameless" representation, with a combination of de Bruijn variables and named free variables, but here we only have pure de Bruijn variables.
- `sort` u represents a universe, written as `Sort` u (or `Prop` and `Type u` which are syntax for `Sort 0` and `Sort (u+1)` respectively).
- `const` c us represents a reference to global constant c in the environment, instantiated with universes us. (Constants in Lean can be universe-polymorphic, and they are instantiated with concrete universes at each use site.)
- `app` f a is function application, written $f\ a$ or $f(a)$.
- `lam` A e is a lambda-expression $\lambda x : A.\, e$. Because we are using de Bruijn variables, x is not represented; instead e is type-checked in an extended context where variable 0 is type A and the other variables are shifted up by 1.
- `forallE` A B[5] is a dependent Pi type $\Pi x : A.\, B$, also written as $\forall x : A.\, B$ because for `Prop` this is the same as the "for all" quantifier. It uses the same binding structure as λ.

The `sort` u and `const` c us constructors depend on another type `VLevel`, which is the grammar of *level expressions*:

```
inductive VLevel where
  | zero  : VLevel
  | succ  : VLevel → VLevel
  | max   : VLevel → VLevel → VLevel
  | imax  : VLevel → VLevel → VLevel -- imax a b means (if b = 0 then 0 else max a b)
  | param : Nat → VLevel
```

These come up when type-checking expressions, for example the type of `sort u` is `sort (succ u)`. The one notable case here is `param` i, which represents the ith universe variable, where $i < n$ where n is the number of universe parameters to the current declaration.

This definition corresponds to the following BNF style grammar from [4], with the main change being that it is more precise about how variables and binding are represented. (For presentational reasons, we will use named variables in the informal version, but the

[4] The prefix "`V`" for "verified" disambiguates it from the `Expr` type used by the kernel itself. This is the more abstracted version of expressions used for specification purposes.
[5] The "`E`" suffix is used because `forall` is a keyword; we could use `forall` anyway but it would require escaping in several places.

$$\boxed{\Gamma \ni x : \alpha} \qquad \textsc{L-ZERO}\ \ \Gamma, x : \alpha \ni x : \alpha \qquad \textsc{L-SUCC}\ \ \frac{\Gamma \ni y : \beta}{\Gamma, x : \alpha \ni y : \beta}$$

$$\boxed{\Gamma \vdash_{E,n} e \equiv e' : \alpha} \qquad (\Gamma \vdash e : \alpha) \triangleq (\Gamma \vdash e \equiv e : \alpha)$$

$$\textsc{T-BVAR}\ \ \frac{\Gamma \ni x : \alpha}{\Gamma \vdash x : \alpha} \qquad \textsc{T-SYMM}\ \ \frac{\Gamma \vdash e \equiv e' : \alpha}{\Gamma \vdash e' \equiv e : \alpha} \qquad \textsc{T-TRANS}\ \ \frac{\Gamma \vdash e_1 \equiv e_2 : \alpha \quad \Gamma \vdash e_2 \equiv e_3 : \alpha}{\Gamma \vdash e_1 \equiv e_3 : \alpha}$$

$$\textsc{T-SORT}\ \ \frac{n \vdash \ell, \ell'\ \mathsf{ok} \quad \ell \equiv \ell'}{\Gamma \vdash \mathsf{U}_\ell \equiv \mathsf{U}_{\ell'} : \mathsf{U}_{S\ell}} \qquad \textsc{T-CONST}\ \ \frac{\bar{u}.(c_{\bar{u}} : \alpha) \in E \quad \forall i,\ n \vdash \ell_i, \ell'_i\ \mathsf{ok}\ \wedge\ \ell_i \equiv \ell'_i}{\Gamma \vdash c_{\bar{\ell}} \equiv c_{\bar{\ell}'} : \alpha[\bar{u} \mapsto \bar{\ell}]}$$

$$\textsc{T-LAM}\ \ \frac{\Gamma \vdash \alpha \equiv \alpha' : \mathsf{U}_\ell \quad \Gamma, x : \alpha \vdash e \equiv e' : \beta}{\Gamma \vdash (\lambda x : \alpha.\ e) \equiv (\lambda x : \alpha'.\ e') : \forall x : \alpha.\ \beta} \qquad \textsc{T-ALL}\ \ \frac{\Gamma \vdash \alpha \equiv \alpha' : \mathsf{U}_{\ell_1} \quad \Gamma, x : \alpha \vdash \beta \equiv \beta' : \mathsf{U}_{\ell_2}}{\Gamma \vdash (\forall x : \alpha.\ \beta) \equiv (\forall x : \alpha'.\ \beta') : \mathsf{U}_{\mathrm{imax}(\ell_1, \ell_2)}}$$

$$\textsc{T-APP}\ \ \frac{\Gamma \vdash e_1 \equiv e'_1 : \forall x : \alpha.\ \beta \quad \Gamma \vdash e_2 \equiv e'_2 : \alpha}{\Gamma \vdash e_1\ e_2 \equiv e'_1\ e'_2 : \beta[x \mapsto e_2]} \qquad \textsc{T-CONV}\ \ \frac{\Gamma \vdash \alpha \equiv \beta : \mathsf{U}_\ell \quad \Gamma \vdash e \equiv e' : \alpha}{\Gamma \vdash e \equiv e' : \beta}$$

$$\textsc{T-BETA}\ \ \frac{\Gamma, x : \alpha \vdash e : \beta \quad \Gamma \vdash e' : \alpha}{\Gamma \vdash (\lambda x : \alpha.\ e)\ e' \equiv e[x \mapsto e'] : \beta[x \mapsto e']} \qquad \textsc{T-ETA}\ \ \frac{\Gamma \vdash e : \forall y : \alpha.\ \beta}{\Gamma \vdash (\lambda x : \alpha.\ e\ x) \equiv e : \forall y : \alpha.\ \beta}$$

$$\textsc{T-PROOF-IRREL}\ \ \frac{\Gamma \vdash p : \mathsf{U}_0 \quad \Gamma \vdash h : p \quad \Gamma \vdash h' : p}{\Gamma \vdash h \equiv h' : p} \qquad \textsc{T-EXTRA}\ \ \frac{\bar{u}.(e \equiv e' : \alpha) \in E \quad \forall i,\ n \vdash \ell_i\ \mathsf{ok}}{\Gamma \vdash e[\bar{u} \mapsto \bar{\ell}] \equiv e'[\bar{u} \mapsto \bar{\ell}] : \alpha[\bar{u} \mapsto \bar{\ell}]}$$

Figure 1 The rules for the judgment $\Gamma \vdash e \equiv e' : \alpha$, with parameters E (the global environment) and n (the number of universe parameters in context), suppressed in the notation. This is `VEnv.IsDefEq` in the formalization.

formalization uses `lift` and `subst` functions when moving expressions between contexts.)

$$\begin{aligned}
\ell &::= 0 \mid S\ell \mid \max(\ell, \ell) \mid \mathrm{imax}(\ell, \ell) \mid u \\
e &::= x \mid \mathsf{U}_\ell \mid c_{\bar{\ell}} \mid e\ e \mid \lambda x : e.\ e \mid \forall x : e.\ e \\
\Gamma &::= \cdot \mid \Gamma, x : e
\end{aligned}$$

2.2 Typing and definitional equality

For the typing rules, we use a slight variation of the typing judgments from [4], see Figure 1. Some remarks:

- In [4], there were two judgments $\Gamma \vdash e : \alpha$ and $\Gamma \vdash e_1 \equiv e_2$ which are mutually inductive (due to T-CONV and T-PROOF-IRREL), and $\Gamma \vdash e_1 \equiv e_2 : \alpha$ was a defined notion for $\Gamma \vdash e_1 \equiv e_2\ \wedge\ \Gamma \vdash e_1 : \alpha\ \wedge\ \Gamma \vdash e_2 : \alpha$. In this version, we have only one all-in-one relation $\Gamma \vdash e_1 \equiv e_2 : \alpha$, and we define $\Gamma \vdash e : \alpha$ to mean $\Gamma \vdash e \equiv e : \alpha$ and define $\Gamma \vdash e_1 \equiv e_2$ as $\exists \alpha.\ (\Gamma \vdash e_1 \equiv e_2 : \alpha)$.

We conjecture the two formulations to be equivalent, but this version seems to be easier to prove basic structural properties about (see subsection 2.3). Moreover Lean does not have good support for mutual inductive predicates, so keeping it as a single inductive makes induction proofs easier.

- We also define $\Gamma \vdash \alpha$ `type` to mean $\exists \ell.\ (\Gamma \vdash \alpha : \mathsf{U}_\ell)$.
- The rules T-SORT, T-CONST, T-EXTRA make use of a judgment $n \vdash \ell$ ok, which simply asserts that every `param` i in ℓ satisfies $i < n$. The $\ell \equiv \ell'$ predicate asserts that ℓ and ℓ' are extensionally equivalent, i.e. for all assignments $v : \mathbb{N} \to \mathbb{N}$ of natural number values to the universe variables, $[\![\ell]\!]_v = [\![\ell']\!]_v$.
- The T-EXTRA rule says that we can add arbitrary definitional equalities from the environment. This is how we will add the two supported "extensions" to the theory, for inductive types and quotients, without the details of these extensions complicating reasoning about the core theory. But we can't actually support *arbitrary* definitional equalities—instead each theorem about the typing judgment has its own assumptions about what extensions are allowed.

2.3 Properties of the typing judgment

▶ **Lemma 1** (basic properties).

1. *(`IsDefEq.closedN`) If $\Gamma \vdash e \equiv e' : \alpha$, then e, e', and α are well-scoped (all free variables have indices less than $|\Gamma|$).*
2. *(`IsDefEq.weakN`, weakening) If $\Gamma, \Gamma' \vdash e \equiv e' : \alpha$, then $\Gamma, \Delta, \Gamma' \vdash e \equiv e' : \alpha$.*
3. *(`IsDefEq.instL`) If $\Gamma \vdash_{E,n} e \equiv e' : \alpha$ and $\forall i.\ n' \vdash \ell_i$, then* $\Gamma[\bar{u} \mapsto \bar{\ell}] \vdash_{E,n'} e[\bar{u} \mapsto \bar{\ell}] \equiv e'[\bar{u} \mapsto \bar{\ell}] : \alpha[\bar{u} \mapsto \bar{\ell}]$.
4. *(`IsDefEq.instN`) If $\Gamma, x : \beta \vdash e_1 \equiv e_2 : \alpha$ and $\Gamma \vdash e_0 : \beta$, then* $\Gamma \vdash e_1[x \mapsto e_0] \equiv e_2[x \mapsto e_0] : \alpha[x \mapsto e_0]$.

Proof sketch.

1. This is a straightforward proof by induction on $\Gamma \vdash e \equiv e' : \alpha$, except that one gets stuck at T-CONST because (assuming the environment is well-typed, see section 4) we know that $\vdash c_{\bar{\ell}} : \alpha$ and need that α is well-scoped. So in fact this is a double induction, first over the environment E and then over $\Gamma \vdash e \equiv e' : \alpha$. We also need lemmas about $e[x \mapsto e']$ and $e[\bar{u} \mapsto \bar{\ell}]$ preserving well-scopedness, but these are also direct by induction on e.
2. By induction. We use Theorem 1.1 in the T-CONST case, to show that the α in $\Gamma, \Gamma' \vdash c_{\bar{\ell}} \equiv c_{\bar{\ell}'} : \alpha$ can be used in $\Gamma, \Delta, \Gamma' \vdash c_{\bar{\ell}} \equiv c_{\bar{\ell}'} : \alpha$ without renaming any bound variables because α is closed.
3. By induction.
4. By induction on the first hypothesis. This uses Theorem 1.2 when lifting $\Gamma \vdash e_0 : \beta$ under binders.

◀

▶ **Lemma 2** (`IsDefEq.defeqDF_l`). *If $\Gamma, x : \alpha \vdash e_1 \equiv e_2 : \beta$ and $\Gamma \vdash \alpha \equiv \alpha' : \mathsf{U}_\ell$, then $\Gamma, x : \alpha' \vdash e_1 \equiv e_2 : \beta$.*

Proof. We have $\Gamma, x : \alpha' \vdash x : \alpha$ by T-BVAR, T-CONV and weakening, and $\Gamma, \_ : \alpha', x : \alpha \vdash e_1 \equiv e_2 : \beta$ by weakening, so $\Gamma, x : \alpha' \vdash e_1[x \mapsto x] \equiv e_2[x \mapsto x] : \beta[x \mapsto x]$ by Theorem 1.4. ◀

▶ **Lemma 3** (`IsDefEq.forallE_inv`). *If $\Gamma \vdash (\forall x : \alpha.\ \beta) : \gamma$ then $\Gamma \vdash \alpha$ `type` and $\Gamma, x : \alpha \vdash \beta$ `type` (and therefore also $\Gamma \vdash (\forall x : \alpha.\ \beta)$ `type`).*

Proof sketch. By induction on $\Gamma \vdash e_1 \equiv e_2 : \gamma$, assuming that one of e_1 or e_2 is $\forall x : \alpha.\ \beta$. Most cases are trivial or inapplicable. Of those that remain:

- T-ALL: If e_1 is $\forall x : \alpha.\ \beta$ we are done; if e_2 is $\forall x : \alpha.\ \beta$ then we obtain $\Gamma, x : \alpha' \vdash \beta$ type and need Theorem 2 to get $\Gamma, x : \alpha \vdash \beta$ type.
- T-BETA: For this to apply, it must be that we have $(\lambda y : \delta.\ e)\ e' \equiv e[y \mapsto e']$ where $e[y \mapsto e']$ is $\forall x : \alpha.\ \beta$. So either $e = y$ and $e' = \forall x : \alpha.\ \beta$, in which case the inductive hypothesis for e' applies, or $e = \forall x : \alpha'.\ \beta'$ with $\alpha'[y \mapsto e'] = \alpha$ and $\beta'[y \mapsto e'] = \beta$ in which case $\Gamma, y : \delta \vdash \alpha'$ type and $\Gamma, y : \delta, x : \alpha' \vdash \beta'$ type by the inductive hypothesis for e, and Theorem 1.4 applies.
- T-EXTRA: It could be that $e[\bar{u} \mapsto \bar{\ell}]$ is $\forall x : \alpha.\ \beta$, but this can only happen if $e = \forall x : \alpha'.\ \beta'$ with $\alpha'[\bar{u} \mapsto \bar{\ell}] = \alpha$ and $\beta'[\bar{u} \mapsto \bar{\ell}] = \beta$, so by induction hypothesis (for the environment) $\vdash \alpha'$ type and $x : \alpha' \vdash \beta'$ type, and we conclude using level substitution and weakening. ◀

▶ **Lemma 4** (`IsDefEq.sort_inv`). *If $\Gamma \vdash_{E,n} \mathsf{U}_\ell : \gamma$ then $n \vdash \ell$ ok (and therefore also $\Gamma \vdash \mathsf{U}_\ell : \mathsf{U}_{S\ell}$).*

Proof sketch. Similar to Theorem 3. ◀

▶ **Theorem 5** (`IsDefEq.isType`). *If $\Gamma \vdash e : \alpha$ then $\Gamma \vdash \alpha$ type.*

Proof sketch. By induction, using weakening and substitution lemmas. The nontrivial cases are:

- T-APP: We have $\Gamma \vdash (\forall x : \alpha.\ \beta)$ type from the IH and $\Gamma \vdash e_2 : \alpha$ by assumption, and from Theorem 3 we get $\Gamma, x : \alpha \vdash \beta$ type, so $\Gamma \vdash \beta[x \mapsto e_2]$ type by the substitution lemma.
- T-ALL: We have $\Gamma \vdash \mathsf{U}_{\ell_1}$ type and $\Gamma, x : \alpha \vdash \mathsf{U}_{\ell_1}$ type from the IH, so by Theorem 4 we have $n \vdash \ell_1, \ell_2$ ok, therefore $\Gamma \vdash \mathsf{U}_{\mathrm{imax}(\ell_1,\ell_2)}$ type.

◀

▶ **Lemma 6** (`IsDefEq.instDF`). *If $\Gamma, x : \alpha \vdash f \equiv f' : \beta$ and $\Gamma, x : \alpha \vdash a \equiv a' : \alpha$ then $\Gamma \vdash f[x \mapsto a] \equiv f'[x \mapsto a'] : \beta[x \mapsto a]$.*

Proof sketch. First we show the claim assuming $\Gamma \vdash \beta[x \mapsto a] \equiv \beta[x \mapsto a'] : \mathsf{U}_\ell$. In this case we have

$$\begin{aligned} f[x \mapsto a] &\equiv (\lambda x : \alpha.\ f)\ a \\ &\equiv (\lambda x : \alpha.\ f')\ a' \\ &\equiv f'[x \mapsto a'] \qquad : \beta[x \mapsto a], \end{aligned}$$

where we use T-BETA twice, and use the assumption $\beta[x \mapsto a] \equiv \beta[x \mapsto a']$ to justify the last step ecause T-BETA gives the equality at the type $\beta[x \mapsto a']$ instead.

To finish, we apply the lemma twice, once with f and β so that it suffices to show $\beta[x \mapsto a] \equiv \beta[x \mapsto a'] : \mathsf{U}_\ell$ and then again with β in place of f and U_ℓ in place of β so that it suffices to show $\mathsf{U}_\ell[x \mapsto a] \equiv \mathsf{U}_\ell[x \mapsto a'] : \mathsf{U}_{S\ell}$, which is true by reflexivity. ◀

2.4 Conjectured properties of the typing judgment

Unfortunately, there are more structural properties of the typing judgment beyond the results in the previous section. Many of these are close relatives of each other and can be proved from other conjectures in this section.

▶ **Conjecture 7** (Unique typing). *If $\Gamma \vdash e : \alpha$ and $\Gamma \vdash e : \beta$, then $\Gamma \vdash \alpha \equiv \beta : \mathsf{U}_\ell$ for some ℓ.*

This has a major simplifying effect on the theory. Here are some consequences (with proofs omitted, but they are simple algebraic consequences of earlier lemmas):

▶ **Corollary 8** (Consequences of unique typing).

1. *(`IsDefEqU.trans`) If $\Gamma \vdash e_1 \equiv e_2$ and $\Gamma \vdash e_2 \equiv e_3$, then $\Gamma \vdash e_1 \equiv e_3$.*
   (This is T-TRANS but without requiring the types to match.)
2. *(`isDefEq_iff`) $\Gamma \vdash e \equiv e' : \alpha$ if and only if $\Gamma \vdash e : \alpha$, $\Gamma \vdash e' : \alpha$, and $\Gamma \vdash e \equiv e'$.*
   (Hence this formulation implies the one of [4].)
3. *(`IsDefEqU.defeqDF`) If $\Gamma \vdash e \equiv e' : \alpha$ and $\Gamma \vdash \alpha \equiv \beta$, then $\Gamma \vdash e \equiv e' : \beta$.*
   (This is T-CONV but with an untyped definitional equality.)

This group of theorems is proved mutually with Theorem 7 in [4]:

▶ **Conjecture 9** (Definitional inversion).

1. *(`IsDefEqU.sort_inv`) If $\Gamma \vdash \mathsf{U}_\ell \equiv \mathsf{U}_{\ell'}$ then $\ell \equiv \ell'$.*
2. *(`IsDefEqU.forallE_inv`)*
   If $\Gamma \vdash (\forall x : \alpha.\ \beta) \equiv (\forall x : \alpha'.\ \beta')$ then $\Gamma \vdash \alpha \equiv \alpha'$ and $\Gamma, x : \alpha \vdash \beta \equiv \beta'$.
3. *(`IsDefEqU.sort_forallE_inv`) $\Gamma \vdash \mathsf{U}_\ell \not\equiv (\forall x : \alpha.\ \beta)$.*

The reason Theorem 7 and Theorem 9 have been downgraded from theorems in [4] to conjectures here is because the proof has an error in one of the technical lemmas, and it remains to be seen if it is possible to salvage the proof. More precisely, the proof constructs a stratification $\vdash_i$ of the typing judgment in order to break the mutual induction between typing and definitional equality, but this stratification does not and cannot respect substitution; that is, if $\Gamma, x : \beta \vdash_i e : \alpha$ and $\Gamma \vdash_j e' : \beta$, then the proof requires $\Gamma \vdash_{\max(i,j)} e[x \mapsto e'] : \alpha$ but only $\Gamma \vdash_{i+j} e[x \mapsto e'] : \alpha$ holds.

We still have reasons to believe the conjectures here are true, but more work is needed to determine how to structure the proof by induction. The proof of soundness is not impacted because there are alternative routes to construct the model that avoid unique typing (also described in [4]), but these conjectures are necessary in at least some form in order to prove the correctness of the typechecker (see section 3).

Another class of conjectured theorems concerns the "invertibility" of weakening:

▶ **Conjecture 10** (Reverse weakening). *If e, e' do not mention any variables in Δ, then $\Gamma, \Delta, \Gamma' \vdash e \equiv e'$ if and only if $\Gamma, \Gamma' \vdash e \equiv e'$.*

▶ **Corollary 11** (Consequences of reverse weakening).

1. *If $\Gamma, \Delta, \Gamma' \vdash e \equiv e' : \alpha$ and e, e' do not mention the variables in Δ, then there exists α' not mentioning Δ such that $\Gamma, \Delta, \Gamma' \vdash e \equiv e' : \alpha'$.*
2. *If e, e', α do not mention any variables in Δ, then $\Gamma, \Delta, \Gamma' \vdash e \equiv e' : \alpha$ if and only if $\Gamma, \Gamma' \vdash e \equiv e' : \alpha$.*
3. *If $\vdash \Gamma, \Delta, \Gamma'$ `ok` and Γ' does not mention the variables in Δ then $\vdash \Gamma, \Gamma'$ `ok`.*

Note that in the formalization, the "e does not mention the variables in Δ" clauses are expressed by saying that e is the `lift` of some e' in the smaller context, because of the use of de Bruijn variables.

3 The typechecker

3.1 Relating VExpr to Expr

The actual Lean kernel does not use any of the machinery from the previous section directly. Instead, it works with another type, `Expr`:

```
inductive Expr where
  | bvar (deBruijnIndex : Nat)
  | fvar (fvarId : FVarId)
  | mvar (mvarId : MVarId)
  | sort (u : Level)
  | const (declName : Name) (us : List Level)
  | app (fn arg : Expr)
  | lam (binderName : Name) (binderType body : Expr) (binderInfo : BinderInfo)
  | forallE (binderName : Name) (binderType body : Expr) (binderInfo : BinderInfo)
  | letE (declName : Name) (type value body : Expr) (nonDep : Bool)
  | lit : Literal → Expr
  | mdata (data : MData) (expr : Expr)
  | proj (typeName : Name) (idx : Nat) (struct : Expr)
```

Comparing this with `VExpr` reveals some differences:

- `mvar` n is for metavariables, which are used for incomplete proofs in the elaborator. It does not matter much for us because the kernel does not create them or allow them in terms submitted to it.
- `fvar` n is a "free variable", which is identified by a name rather than as an index which changes depending on the context. Although these are also not allowed in terms provided to the kernel, it does create `fvar` n expressions during typechecking, and converting between locally nameless and pure de Bruijn is one of the major differences we will have to deal with in verification of the kernel.
- The `Level` type (not shown) used in the `sort` constructor also differs in similar ways to `VLevel`, containing a `mvar` constructor we don't care about and a `param` constructor which takes a `Name` instead of a `Nat`.
- The `lam` and `forallE` constructors contain information which is relevant only for elaboration and printing like the names of bound variables and whether the argument is implicit or explicit.
- The `letE` construct is used for `let x := e1; e2` expressions. We handle these by simply expanding them to $e_2[x \mapsto e_1]$ in the abstract theory, meaning that we don't have to handle this constructor while doing induction proofs about the typing judgment. The kernel puts these in the context and unfolds them as necessary.
- The `lit` constructor is for literals:

  ```
  inductive Literal where
    | natVal (val : Nat)
    | strVal (val : String)
  ```

  This is one of the new features of the Lean 4 kernel: natural numbers are represented directly as bignums and several functions like `Nat.mul` are overridden with a native implementation rather than using the inductive structure of `Nat` directly (effectively writing numerals in unary). For the purpose of modeling in `VExpr` however, we just perform exactly this (exponential-size) unfolding in terms of `Nat.zero` and `Nat.succ` applications. Similarly, `String` is a wrapper around `List Char` for the purpose of modeling, even though the actual representation in the runtime is as a UTF-8 encoded byte string.
- The `mdata` d e constructor is equivalent to e for modeling purposes (this is used for associating metadata to expressions). We can't completely ignore it though because it has a different hash than e, which impacts some of the caches used in the kernel.
- `proj` c i e is a projection out of a `structure` or structure-like inductive type. (See section 5.) This is the most challenging constructor to desugar, because even though it is equivalent

to an application of the recursor for the inductive type, we need the type of e to construct this term. This means that the desugaring function must be type-aware, and is only well-defined up to definitional equality.

It is worth mentioning that `Expr` was not defined as part of this project — this is a type imported directly from the Lean elaborator. In fact, the C++ Lean kernel uses FFI (foreign-function interface) to Lean in order to make use of this type, so Lean4Lean and the C++ kernel are sharing this data structure.

As a result of these considerations, we use an inductive type to encapsulate the translation from `Expr` to `VExpr`:

```
inductive TrExpr (env : VEnv) (Us : List Name) : VLCtx → Expr → VExpr → Prop
```

Here `TrExpr` env Us Δ e e' asserts that in the local context Δ, with names Us for the local universe parameters and env for the global environment (see section 4), e' `: VExpr` is a translation of e' `: Expr`. It also asserts that Δ is a well-typed context and e' is well-typed in it, so we can use $\exists e'$. `TrExpr` env Us Δ e e' to assert that e is a well typed `Expr`, which is the target specification we have for correctness of the verifier.

The type `VLCtx` appearing here is new, and it contains the information we need to translate between the `Expr` and `VExpr` types:

```
inductive VLocalDecl where
  | vlam (type : VExpr)          -- x : type
  | vlet (type value : VExpr)    -- x : type := value

def VLCtx := List (Option FVarId × VLocalDecl)
```

The idea is that `vlam` α represents a regular variable in the context, the only kind of variable we had in section 2, while `vlet` α v is used when we are inside the context of a `let x :` α `:=` v`; _` term. We can build a local context in the sense of section 2 by simply dropping the `vlet` α v terms:

```
def toCtx : VLCtx → List VExpr
  | [] => []
  | (_, vlam α) :: Δ => α :: toCtx Δ
  | (_, vlet _ _) :: Δ => toCtx Δ
```

We have to be careful to reindex the `Expr.bvar` i indices though, since this includes both let- and lambda-variables while `VExpr.bvar` i only counts the lambda-variables, with the let-variables expanded to terms.

The `Option FVarId` in the context represents the "name" of the variables. Lean follows the "locally nameless" discipline, in which most operations act only on "closed terms" (where closed means that there are no `bvar` i variables outside a binder), so any time it needs to process a term under a binder it first *instantiates* the term, replacing `bvar` $(i+1)$ with `bvar` i and `bvar 0` with `fvar` a, where a `: FVarId` is a (globally) fresh variable name. We store these names in Δ so that we know how to relate them to de Bruijn indices in the context.

The reason it is an `Option` is because when we are typing an *open* term, we still have to deal with variables that have not been assigned `FVarId`s. Even though Lean does not directly handle such terms, they appear as subterms of expressions that are handled so we need both cases to have a compositional specification.

3.2 The typechecker implementation

The kernel itself is defined completely separately from `VExpr`, and is a close mirror of the C++ code, modulo translation into functional style. For example, the code for inferring/checking the type of $\lambda x : \alpha.\ e$:

```
-- Lean4Lean
def inferLambda (e : Expr) (inferOnly : Bool) : RecM Expr := loop #[] e where
  loop fvars : Expr → RecM Expr
  | .lam name dom body bi => do
    let d := dom.instantiateRev fvars
    let id := ⟨← mkFreshId⟩
    withLCtx ((← getLCtx).mkLocalDecl id name d bi) do
      let fvars := fvars.push (.fvar id)
      if !inferOnly then
        _ ← ensureSortCore (← inferType d inferOnly) d
      loop fvars body
  | e => do
    let r ← inferType (e.instantiateRev fvars) inferOnly
    let r := r.cheapBetaReduce
    return (← getLCtx).mkForall fvars r
```

```
// C++ Lean kernel
expr type_checker::infer_lambda(expr const & _e, bool infer_only) {
  flet<local_ctx> save_lctx(m_lctx, m_lctx);
  buffer<expr> fvars;
  expr e = _e;
  while (is_lambda(e)) {
    expr d = instantiate_rev(binding_domain(e), fvars.size(), fvars.data());
    expr fvar = m_lctx.mk_local_decl(m_st->m_ngen, binding_name(e), d, binding_info(e));
    fvars.push_back(fvar);
    if (!infer_only) {
      ensure_sort_core(infer_type_core(d, infer_only), d);
    }
    e = binding_body(e);
  }
  expr r = infer_type_core(instantiate_rev(e, fvars.size(), fvars.data()), infer_only);
  r = cheap_beta_reduce(r);
  return m_lctx.mk_pi(fvars, r);
}
```

Although control flow is expressed differently (here, the replacement of a `while` loop with a tail-recursive function), the two implementations do essentially the same things in the same order. As a result, most bugs or peculiarities we have discovered in the course of formalizing Lean4Lean are replicated in the C++ kernel, which gives us hope that verifying Lean4Lean will lend strong evidence to the correctness of the C++ code as well.

3.2.1 Termination and the RecM monad

The function above is defined in the `RecM` monad, which is defined like so:

```
structure Methods where
  isDefEqCore : Expr → Expr → M Bool
  whnfCore (e : Expr) (cheapRec := false) (cheapProj := false) : M Expr
  whnf (e : Expr) : M Expr
  inferType (e : Expr) (inferOnly : Bool) : M Expr

abbrev RecM := ReaderT Methods M
```

This is a monad stack on top of the main `TypeChecker.M` monad which contains the actual state of the typechecker. The use of `Methods` here is a trick for cutting the knot of mutual definitions involved in the kernel: removing calls to these four functions makes the call graph acyclic (except for self-loops corresponding to directly recursive definitions). So we can avoid mutual definitions — every definition only refers to itself and earlier functions.

There are a few reasons to avoid mutual definitions:

- Lean is bad at elaborating large mutual blocks, because it cannot process them sequentially. This is not critical but it makes working on the code more difficult.
- Mutual definitions generally compile to a well founded recursion, while we want to utilize structural recursion when possible because of the better induction principles.
- Theorems about mutual definitions also have to be mutual, so this has a tendency toward lack of modularity.
- As we will see, there is no suitable well founded measure in the first place, so we would be forced to declare the definition as `partial`, but we absolutely cannot do this if we want any hope of proving correctness because it would make our functions opaque, so we need another option.

Having a specific list of cut points for the graph of mutual definitions doesn't make them any less recursive, but at least gives us a clear structure in which to perform the proof. The four functions above were found by computer search, but they are sensible functions with a clear spec so it is not too surprising that they are used throughout the kernel:

- `isDefEqCore` s t returns `true` if $\Gamma \vdash s \equiv t$ is provable. (This is normally used via `isDefEq` s t, which also caches this result, but some callers use `isDefEqCore` directly.)
- `whnf` e returns the weak head normal form (WHNF) of e. From a modeling perspective, the main important property is that if it returns e' and e is well-typed then $\Gamma \vdash e \equiv e'$ is provable.
- `whnfCore` e `cheapRec` `cheapProj` has the same specification as `whnf` e. `cheapRec` and `cheapProj` are flags affecting which kinds of terms are unfolded. The reason both `whnf` and `whnfCore` show up separately in this list is because they are both used in other functions.
- `inferType` e `inferOnly` infers or type-checks a term. That is, if `inferType` e `inferOnly` returns α then $\Gamma \vdash e : \alpha$ holds if either e is well typed or `inferOnly` is false.

Now, this method of calling functions to avoid recursion only kicks the problem up one step. How do we construct an element of `Methods` if all the functions require another `Methods`? Ideally, we would actually prove the termination of the kernel, because DTT is supposed to be terminating. However:

- It is unlikely that we can prove termination of a typechecker for Lean in Lean, because although our soundness proof does not depend on termination, MetaCoq's does [11], and generally termination measures for DTT require large cardinals of comparable strength to the proof theory. We are up against Gödel's incompleteness theorem, so anything that would imply the unconditional soundness of Lean won't be directly provable.
- Besides this, the Lean type theory is known not to terminate. Coquand and Abel [1] constructed a counterexample to strong normalization using reduction of proofs, and this can be shown to impact definitional equality checks even for regular types:

```
/-! Andreas-Abel construction of nontermination in proofs -/
def True' := ∀ p : Prop, p → p
def om : True' := fun A a =>
   @cast (True' → True') A (propext ⟨fun _ => a, fun _ => id⟩) <|
   fun z => z (True' → True') id z
def Om : True' := om (True' → True') id om
#reduce Om -- whnf nontermination

/-! nontermination outside proofs: -/
inductive Foo : Prop | mk : True' → Foo
def foo : Foo := Om _ (Foo.mk fun _ => id)
example : foo.recOn (fun _ => 1) = 1 := by rfl -- isDefEq nontermination
```

Essentially, this is a combination of impredicativity, proof irrelevance, and subsingleton elimination.

- The kernel does not loop forever in many cases because this is a bad user experience — it has timeouts and depth limits. Not all parts of the kernel have such limits, but it does give us a reasonable design principle which fortuitously solves our termination problem.

So we use what is arguably the standard solution for defining partial functions in a language like Lean or Coq: use a fuel parameter, a natural number which counts the number of nested recursive calls to one of the `Methods`, and throw a `deepRecursion` error if we run out of fuel. Currently, this limit is a fixed constant (1000), which turns out to be sufficient for checking all of `Mathlib`, but this could be made configurable. (Note that this is not a limit on the depth of expressions exactly, these use structural recursion and hence need no fuel for the termination argument; instead fuel is only consumed when making a recursive call that is not otherwise decreasing, for example when reducing definitions to WHNF. The code that is most likely to hit depth limits is in proofs by reflection, but these are comparatively rare, in part because the kernel algorithm for this is not very efficient.)

4 The global environment

The environment is the global structure that ties together individual declarations. It has made some appearances in the previous sections already, because the environment is needed to typecheck constants as well as definitional extensions, given in the T-CONST and T-EXTRA rules.

The Lean type for this, `Environment`, is complex and contains many details irrelevant to the kernel, but luckily we only really care about a few operations on it:

- `add : Environment → ConstantInfo → Environment` – add a `ConstantInfo` declaration to the environment
- `mkEmptyEnvironment : IO Environment` – constructs an environment with no constants[6]
- `find? : Environment → Name → Option ConstantInfo` – retrieves a declaration by name

In effect, the environment is just a fancy hashmap which indexes `ConstantInfo` declarations by their names.

The corresponding theory type is called `VEnv`, with the definition:

```
structure VConstant where (uvars : Nat) (type : VExpr)
structure VDefEq where (uvars : Nat) (lhs rhs type : VExpr)
structure VEnv where
```

[6] This is in `IO` because environment extensions can perform computation in `IO` for their initialization. But the constant map, which is what we care to verify, is initialized to the empty map.

```
  constants : Name → Option (Option VConstant)
  defeqs : VDefEq → Prop
```

This is a very simple type, essentially exactly what is needed to satisfy the requirements of the typing judgment.

- `constants` maps a name to a constant, if defined, where a constant is given by its universe arity and its type. It returns `none` if a constant by that name does not exist, and `some none` if the name has been "blocked", meaning that there is no constant there but we still want to make it illegal to make a definition with that name. This is how we model `unsafe` declarations when typechecking safe declarations: they "don't exist" for most purposes, but it is nevertheless not allowed to shadow them with another safe declaration. This corresponds to the $\bar{u}.(c_{\bar{u}} : \alpha)$ declaration appearing in T-CONST.
- `defeqs` is a set of (anonymous) `VDefEq`s, containing two expressions to be made definitionally equal and their type. This corresponds to the $\bar{u}.(e \equiv e' : \alpha)$ declaration appearing in T-EXTRA.

In some sense this doesn't actually answer many interesting questions about where the constants come from or what definitional equalities are permitted. For that, we need `VDecl`, the type of records that can be used to update the environment. The declarations are:

- `block` n blocks constant n, setting `env.constants` n `= some none` and preventing a redeclaration as mentioned above.
- `axiom { name :=` c`, uvars :=` n`, type :=` α `}` adds $c : \alpha$ as an axiom to the constant map.
- `opaque { name :=` c`, uvars :=` n`, type :=` α`, value :=` e `}` checks that $e : \alpha$, then adds $c : \alpha$ to the constant map.
- `def { name :=` c`, uvars :=` n`, type :=` α`, value :=` e `}` does the same as `opaque` but also adds $c \equiv e : \alpha$ to the defeq set.
- `example { uvars :=` n`, type :=` α`, value :=` e `}` checks that $e : \alpha$ and then leaves the environment as is.
- `induct (`d `: VInductDecl)` adds all the constants and defeqs from an inductive declaration (see section 5).
- `quot` adds the quotient axioms and definitional equality. This is a type operator α/R where $R : \alpha \to \alpha \to$ `Prop` with $\mathrm{mk} : \alpha \to \alpha/R$ and $\mathrm{lift} : (f : \alpha \to \beta) \to (\forall xy.\ R\ x\ y \to f\ x = f\ y) \to (\alpha/R \to \beta)$, with the property that $(\mathrm{lift}\ f\ h\ (\mathrm{mk}\ a) \equiv f\ a : \beta)$. (One can construct every part of this in Lean except for the definitional equality.)

This enumeration is fairly similar to the `Declaration` type which is the actual front end to the kernel, but it lacks `unsafe` declarations and `mutualDefnDecl` (which, despite the name, is not the way mutual definitions are sent to the kernel, but rather represents unsafe declarations with unchecked self-referential definitions). For the most part we do not attempt to model `unsafe` declarations because these are typechecked with certain checks disabled, and this makes them no longer follow the theory, but this is deliberate. There isn't much we can say about such definitions since they can violate type safety and cause undefined behavior, and in any case they don't play a role in checking theorems.

5 Inductive types

Lean's implementation of inductive types is based on Dybjer [8]. It notably differs from Coq in that rather than having primitive `fix` and `match` expressions, we have constants called "recursors" generated from the inductive specification. This choice significantly simplifies the `VExpr` type, which as we have seen contains no special support for anything inductive-related except for the generic def.eq. hook T-EXTRA. It also means we do not need a complex guard checker, something which MetaCoq currently axiomatizes ([11]) and which has been the source of soundness bugs in the past. Instead, the complexity is pushed to the generation of the recursor for the inductive type.

For single inductives, the algorithm is described in full in [4]. In Lean 3, nested and mutual inductives were simulated using single inductives so the kernel only had to deal with the single inductive case. However this simulation process was both costly and did not cover the full grammar of nested and mutual inductives, so Lean 4 moved the checking to the kernel.

For a simple example of a mutual inductive, we can define a data-carrying version of `Even` and `Odd` by mutual induction like so:

```
mutual
inductive Even : Nat → Prop where
  | zero : Even 0
  | succ : Odd n → Even (n+1)

inductive Odd : Nat → Prop where
  | succ' : Even n → Odd (n+1)
end
```

which generates the definitions:

```
Even : Nat → Type
Even.zero : Even 0
Even.succ : ∀ {n : Nat}, Odd n → Even (n + 1)
Odd : Nat → Type
Odd.succ' : ∀ {n : Nat}, Even n → Odd (n + 1)
Even.rec.{u} :
  {motive_1 : (a : Nat) → Even a → Sort u} →
  {motive_2 : (a : Nat) → Odd a → Sort u} →
  motive_1 0 Even.zero →
  ({n : Nat} → (a : Odd n) → motive_2 n a → motive_1 (n + 1) (Even.succ a)) →
  ({n : Nat} → (a : Even n) → motive_1 n a → motive_2 (n + 1) (Odd.succ' a)) →
  {a : Nat} → (t : Even a) → motive_1 a t
Odd.rec.{u} :
  ... → -- same as Even.rec
  {a : Nat} → (t : Odd a) → motive_2 a t
```

and the definitional equalities:

```
@Even.rec motive_1 motive_2 F_zero F_succ F_succ' 0 Even.zero ≡ F_zero
@Even.rec motive_1 motive_2 F_zero F_succ F_succ' (n+1) (Even.succ e) ≡
  @F_succ n e (@Odd.rec motive_1 motive_2 F_zero F_succ F_succ' n e)
@Odd.rec motive_1 motive_2 F_zero F_succ F_succ' (n+1) (Odd.succ' e) ≡
  @F_succ' n e (@Even.rec motive_1 motive_2 F_zero F_succ F_succ' n e)
```

For nested inductives, the checking procedure effectively amounts to running the simulation procedure from Lean 3 to get a mutual inductive type and performing well-formedness checks there, then removing all references to the unfolded type in the recursor. For example:

```
inductive T : Type where
  | mk : List T → T
```

```
T.rec.{u} :
  {motive_1 : T → Sort u} → {motive_2 : List T → Sort u} →
  ((a : List T) → motive_2 a → motive_1 (T.mk a)) →
  motive_2 [] →
  ((head : T) → (tail : List T) →
    motive_1 head → motive_2 tail → motive_2 (head :: tail)) →
  (t : T) → motive_1 t
```

This recursor (there is also `T.rec_1`, not shown) appears exactly like one would expect for a mutual inductive with `T` and `List_T` defined the same way as `List`, except that all occurrences of `List_T` in the recursor type have been removed in favor of `List T`.

Currently, `Lean4Lean` contains a complete implementation of inductive types, including nested and mutual inductives and all the type-checking effects of this, but not much work has been done on the theoretical side, defining what an inductive specification should generate. Ideally we would like to do it in a way which is not so intimately tailored to the nested inductive collection process (i.e. we can see the order in which subterms are visited by the order of `motive_`n arguments in the above recursor) and instead allows a range of implementation strategies which can all be proved sound against a set-theoretic model.

5.1 Eta for structures

Another new kernel feature in Lean 4 is primitive projections, and eta for structures, which says that if s is a `structure` (which is to say, a one-constructor non-recursive `inductive` type), then $s \equiv \mathrm{mk}(s.1, \dots, s.n)$ where $s.i$ is the i'th projection (`Expr.proj`). This was already true as a propositional equality proven by induction on s, but it is now a definitional equality and the kernel has to unfold structures when needed to support this. This does not fundamentally change the theory; indeed in [4] a mild form of eta for pairs was introduced in order to make the reduction to W-types work, so we believe that the main results should still hold even with this extension.

6 Results

`Lean4Lean` contains a command-line frontend, which can be used to validate already-compiled Lean projects. It operates on `.olean` files, which are essentially serialized (unordered) lists of `ConstantInfo` objects, and the frontend topologically sorts these definitions and passes them to the `Lean4Lean` kernel, which is a function

```
def addDecl' (env : Environment) (decl : Declaration) (check := true) :
  Except KernelException Environment
```

that is a drop-in replacement for `Environment.addDecl`, which is an opaque Lean function implemented by FFI (foreign-function interface) to the corresponding C++ kernel function.

This is the same interface (and to some extent, the same code) as `lean4checker`[7], which does the exact same thing but calls `addDecl` instead, effectively using the C++ kernel as an external verifier for Lean proofs. It may seem odd to use the Lean kernel as an external verifier for itself, but this can catch cases where malicious (or confused) Lean code makes use of the very powerful metaprogramming framework to simply bypass the kernel and add constants to the environment without typechecking them.

[7] https://github.com/leanprover/lean4checker

	`lean4export`	`lean4lean`	ratio
`Lean`	37.01 s	44.61 s	1.21
`Std`	32.49 s	45.74 s	1.40
`Mathlib` (+ `Std` + `Lean`)	44.54 min	58.79 min	1.32

Figure 2 Comparison of the original C++ implementation of the Lean kernel (`lean4export`) with `lean4lean` (Lean) on major Lean packages. Tests were performed on a 12 core 12th Gen Intel i7-1255U @ 2.1 GHz, single-threaded, on rev. 526c94c of `mathlib4`.

For our purposes, it acts as a very handy benchmark for comparison, since we are performing the same operations but with a different kernel. Running `lake env lean4lean --fresh Mathlib` in the `mathlib4` project will check the `Mathlib` module and all of its dependencies. (This is running in single-threaded mode; it can also check modules individually, assuming the correctness of imports, but it runs into memory usage limitations so is harder to benchmark reliably.)

The results are summarized in Figure 2. The new Lean implementation is around 30% slower than the C++ implementation, which we attribute mainly to shortcomings in the Lean compiler and data representation compared to C++. Nevertheless, it is within an order of magnitude and practically usable on large libraries, which we consider a strong sign.

7 Related work & Conclusion

Self-verification of ITP systems has been done before. The most relevant references are [2] for Coq, [9] for HOL Light, [5] for Metamath Zero, and to some extent self-verification also overlaps with bootstrapping theorem provers such as Milawa [6] and CakeML [10]. But the most similar work is unquestionably MetaCoq [11], which is a project to develop a verified type-checker for Coq, in Coq. It is significantly more complete than the previous effort [2], but is not yet capable of verifying the Coq standard library because it does not support some of the more unusual or sketchy (mis)features in Coq and there is some effort to get these features removed rather than attempt to validate them. With Lean4Lean we took the approach of quickly getting to feature parity with the real Lean kernel and proving correctness later, so we ended up with a complete verifier but are still lacking in some theorems.

As was mentioned, MetaCoq also has to deal with many of the same issues proving properties of the typing judgment, and one may hope for proofs from MetaCoq to be useful in Lean4Lean and vice-versa. Unfortunately, there are some important details that differ:

- In MetaCoq, the conversion relation is a partial order rather than an equivalence relation, because of universe cumulativity, so Theorem 7 doesn't hold as stated. Nevertheless, there are theorems regarding "principal types" which can be used to play the same role.
- Unlike in Lean4Lean, the conversion relation is *untyped*: there is no mutual induction between typing and definitional equality, which massively simplifies matters. In Lean this is unfortunately not an option because of the T-PROOF-IRREL rule, which is absent in Coq.

In this paper we presented a new external verifier for Lean, but it is also the beginning of a larger project to both verify the correctness of a verifier with respect to the Lean metatheory, and also to verify that the Lean metatheory is consistent relative to well-studied axiomatic foundations like ZFC with inaccessible cardinals. Lean 4 is written in Lean, so this is a prime opportunity to have a production-grade ITP which is able to bootstrap at the logical level.

References

1 Andreas Abel and Thierry Coquand. Failure of Normalization in Impredicative Type Theory with Proof-Irrelevant Propositional Equality. *Logical Methods in Computer Science*, Volume 16, Issue 2, June 2020. URL: `https://lmcs.episciences.org/6606`, `doi:10.23638/LMCS-16(2:14)2020`.

2 Bruno Barras and Benjamin Werner. Coq in Coq. *Available on the WWW*, 1997.

3 Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Springer Science & Business Media, 2013.

4 Mario Carneiro. The Type Theory of Lean. 2019. URL: `https://github.com/digama0/lean-type-theory/releases/tag/v1.0`.

5 Mario Carneiro. Metamath zero: Designing a theorem prover prover. In *Intelligent Computer Mathematics: 13th International Conference, CICM 2020, Bertinoro, Italy, July 26–31, 2020, Proceedings*, page 71–88, Berlin, Heidelberg, 2020. Springer-Verlag. `doi:10.1007/978-3-030-53518-6_5`.

6 Jared Curran Davis and J Strother Moore. *A self-verifying theorem prover*. PhD thesis, University of Texas, 2009.

7 Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. The Lean Theorem Prover (system description). In *International Conference on Automated Deduction*, pages 378–388. Springer, 2015.

8 Peter Dybjer. Inductive Families. *Formal aspects of computing*, 6(4):440–465, 1994.

9 John Harrison. Towards self-verification of hol light. In Ulrich Furbach and Natarajan Shankar, editors, *Proceedings of the third International Joint Conference, IJCAR 2006*, volume 4130 of *Lecture Notes in Computer Science*, pages 177–191, Seattle, WA, 2006. Springer-Verlag.

10 Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: A Verified Implementation of ML. *SIGPLAN Not.*, 49(1):179–191, January 2014. URL: `http://doi.acm.org/10.1145/2578855.2535841`, `doi:10.1145/2578855.2535841`.

11 Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. Coq Coq correct! verification of type checking and erasure for Coq, in Coq. *Proc. ACM Program. Lang.*, 4(POPL), dec 2019. `doi:10.1145/3371076`.

12 Sebastian Andreas Ullrich. *An Extensible Theorem Proving Frontend*. PhD thesis, Karlsruher Institut für Technologie (KIT), 2023. `doi:10.5445/IR/1000161074`.