Lean4Lean: Mechanizing the Metatheory of Lean

Mario Carneiro

Chalmers University of Technology
University of Gothenburg
Gothenberg, Sweden
marioc@chalmers.se

Introduction. The Lean proof assistant [2] is seeing increasing use in mathematics formalization and software verification, but the metatheory of Lean is not yet completely understood. In particular, while there is an intended interpretation within a classical set-theoretical model ([1], based on [5]), the proof of soundness has not been formalized. The Lean4Lean project endeavors to verify the key theorems (and non-theorems!) of the LeanTT formal system. Additionally, Lean4Lean provides an executable typechecker, written in Lean itself. This is the first complete typechecker for Lean 4 other than the reference implementation in C++ used by Lean itself, and our new checker 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. Ultimately, 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.

The project is most directly comparable to the MetaRocq project [3, 4], and aims to do the same things that it accomplishes for the Rocq theorem prover, but for Lean's type theory. This is complicated by the fact that LeanTT is known to not be theoretically optimal: in particular, the typing relation is undecidable and Lean underapproximates it, and reduction is not strongly normalizing. (Canonicity also fails for the basic reason that LeanTT includes 3 axioms,[1] one of which is the axiom of choice.)

Project structure. The Lean4Lean project consists of three major components:

- **Executable typechecker.** This is a "carbon copy" reimplementation of the C++ Lean kernel written in Lean. It is written in monadic style, using only pure functions (i.e. using some combination of state, reader and exception monads rather than `IO`), and is designed to be efficiently compiled by the Lean compiler so that the resulting binary is competitive with the original. Because Lean is a functional programming language with limited optimizations, there is still a performance gap of about 20–50% compared to the C++ version, but this may improve along with the Lean compiler. This is already good enough to be usable as an additional or alternative checking step for Lean projects, and we would like to make a case for it to *replace* the C++ kernel.[2]
- **Formalized metatheory.** This is the most interesting part of the project for type theorists, and amounts to a definition of a type `VExpr` representing expressions (extrinsically typed, using pure de Bruijn variable convention), a judgment $\Gamma \vdash e \equiv e' : \tau$ which deals

[1] The kernel actually implements a theory LeanTT_0 which lacks these axioms but this system is not as well studied. LeanTT adds to LeanTT_0 the axioms `propext`, `Quot.sound`, and `choice`, and the main metatheory, including the soundness proof, targets this extended system.

[2] The actual decision for if and when this happens is under the control of the Lean FRO, so I cannot speculate on a timeline here. If the winds are right it could happen tomorrow, but the in-development next generation Lean compiler may help reduce the performance gap. There are also practical considerations regarding maintenance in getting the Lean4Lean kernel to be shipped with Lean.

with typing and definitional equality, and a number of theorems and conjectures[3] about the behavior of this relation. These types are a "cleaned-up" version of the corresponding types used in the executable kernel (which uses the locally-nameless variable convention, and includes many conservative extensions such as number and string literals, and primitive projections), to make the presentation of the theory more natural and simplify the task of proving the metatheorems.

- **Program verification.** Given the previous two components, the natural next step is to combine them, such that the executable kernel can have invariants at each stage, culminating in a theorem to the effect that "if `typecheck` e `= ok` τ, then $\vdash [\![e]\!] : [\![\tau]\!]$".[4]

Formalization progress. The project is available at https://github.com/digama0/lean4lean.

Currently, the most complete portion of the project is the **executable typechecker**, which is completely implemented and can typecheck arbitrary Lean projects. The one unsupported Lean kernel feature is the `reduceBool` function, exposed to users via the `native_decide` tactic, which allows the kernel to use the Lean interpreter to run arbitrary Lean code (and in fact, via `@[extern]` annotations, even truly external code such as C or Python) and trust the result. Luckily, this function is (rightfully) viewed with suspicion by most Lean users, and it is banned in many projects such as Mathlib which do not want the expansion of the trusted code base (TCB) implied by this feature.

The **formalized metatheory** portion of the project is well underway, with all of the main definitions stated: expressions, universe levels, contexts, declarations and environments, with the exception of inductive types, which have only the basic scaffolding. (Already many of the "interesting" features of the type theory arise when considering pure MLTT with a universe hierarchy, impredicative `Prop`, and definitional proof irrelevance.) Well-formedness and typing judgments are layered on top of this substrate, with the main player being the $\Gamma \vdash e \equiv e' : \tau$ judgment, defined as an inductive predicate (subsuming both expression typing and definitional equality).

There are some key missing theorems. The most complex theorem in [1] is the unique typing theorem, which says that if $\Gamma \vdash e : \alpha$ and $\Gamma \vdash e : \beta$ then $\Gamma \vdash \alpha \equiv \beta$. It is proved mutually with injectivity of type constructors (e.g. $\mathsf{U}_\ell \equiv \mathsf{U}_{\ell'}$ implies $\ell \equiv \ell'$) and uses a variation on the Tait–Martin-Löf method for proving the Church–Rosser theorem for a specially defined reduction relation which restores confluence while abandoning termination. This theorem is used directly and indirectly in the justification of some of the optimizations employed in the executable kernel.

The final component of the project, **program verification**, is in its early stages. The $\Gamma \vdash [\![e]\!] \Rightarrow e'$ relation which relates the `Expr` type used by the executable kernel to the `VExpr` type used by the theory is defined and some of its basic properties are established. There are similar interpretation relations for most of the other kernel concepts: `LocalContext`, the analogue of the Γ context in the above relations, `Environment` for global declarations, as well as some other material like `NameGenerator` (which is a global counter which ensures new free variables are fresh) and the expression typing cache. But work on giving actual proofs of

[3] The "conjectures" here refer to theorems from [1] with informal proofs which have yet to be formalized. I cannot say with confidence that there are no errors remaining in the informal proof, as it is a very delicate argument. This is part of the reason for wanting to formalize it in the first place.

[4] In actuality the theorem is quite a bit more complicated than this to state, because the typechecker has many additional inputs, such as the environment which supplies the definitions of constants. The interpretation function $[\![\cdot]\!]$ is also not a function, but a relation, because the translation from `Expr` to `VExpr` is not purely syntactic but depends on typing. But this is the essence of the theorem statement.

(the substantially large) kernel functions has yet to properly begin, and concurrent work on a framework for verification of monadic programs by upstream Lean may make this easier.

Future work. The project is quite ambitious and still far from complete, but there are more goals to aspire to beyond this. We already mentioned that `Lean4Lean` does not support `reduceBool`, but this is both possible and desirable for heavy computations which use this feature for performance. It would require formalizing the Lean intermediate representation and the frontend of the compiler, along with some theorems about erasure as in [4]. Furthermore there are more radical changes to kernel reduction (e.g. NbE) which are currently too risky to consider without verification. There are no concrete proposals for changes to reduction at the moment, largely because it's a complete non-starter under the status quo, but it is widely understood that the kernel is very algorithmically inefficient and has performance issues, and this makes it more difficult to practically use implementation techniques such as proof by reflection, widely used in Rocq, in performance-conscious Lean tactics. But there is a large literature on better type checking strategies and Lean4Lean provides a place to experiment and benchmark alternatives, and possibly compensate for the lost 30% performance difference and then some.

References

[1] Mario Carneiro. The Type Theory of Lean. Master's thesis, Carnegie Mellon University, 2019.

[2] 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.

[3] Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. The MetaCoq Project. *Journal of Automated Reasoning*, February 2020.

[4] 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.

[5] Benjamin Werner. Sets in types, types in sets. In *International Symposium on Theoretical Aspects of Computer Software*, pages 530–546. Springer, 1997.