Lean4Lean: Verifying a Typechecker for Lean, in Lean

Mario Carneiro
Chalmers University of Technology & University of Gothenburg, Sweden

Abstract

In this paper we present a new "external checker" for the Lean theorem prover, written in Lean itself. This is the first[1] 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, taking an additional step in Lean's aim to self-host the full elaborator and compiler. Moreover, because the checker 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 progress to formalize the Lean type theory abstractly and prove some theorems about it. Finally, we combine these to get a proof of correctness of parts of the kernel. 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 verification is already paying off, as one soundness bug has been spotted and fixed as a result of this work.

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 typechecker, implementation, metatheory, type theory, proof theory

Digital Object Identifier 10.4230/LIPIcs.TYPES.2025.2

Related Version *Full Version*: `https://arxiv.org/abs/2403.14064`

Supplementary Material *Software (Formalization & Implementation)*: `https://github.com/digama0/lean4lean/tree/types2025`
archived at `swh:1:dir:02a8cfda4f3f7a47d8e446a4173279d0374d218e`

Acknowledgements I would like to thank Jeremy Avigad for their support and encouragement, Yannick Forster and the anonymous reviewers for reading early drafts of this work, and Matthieu Sozeau and the rest of the MetaRocq team for collaborating and taking interest in this work. Work supported by the Hoskinson Center for Formal Mathematics at Carnegie Mellon.

1 Introduction

Lean [12] is a theorem prover based on the Calculus of Inductive Constructions (CIC) [23], quite similar to its older brother Rocq[2] [31], but as the name suggests, one of the ways it sought to differentiate itself 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 Rocq[3] due to complications of the many interacting features in the kernel. So from the beginning, Lean advertised its simpler foundations and multiple external checkers [25, 14] as reasons that it would not suffer from the same troubles.

[1] Although more checkers such as nanoda have appeared in the meantime.
[2] Formerly known as Coq.
[3] `https://github.com/rocq-prover/rocq/blob/master/dev/doc/critical-bugs.md`

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 prior work *Lean's Type Theory* [7] (henceforth LTT) showed the consistency of Lean with respect to ZFC plus the existence of n inaccessible cardinals for each $n < \omega$, so the soundness story seemed strong.

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 checkers: the whole metaprogramming infrastructure was redesigned so old external checkers 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 in LTT [7] is no longer directly applicable. However, the work by Ullrich [32] covers some of these modifications.

We interpret this as a cautionary tale: It is not sufficient to see a prior system and hope to do better through sheer force of will. We are all human, and mistakes in both the theory and the implementation happen as a matter of course. The way to do better is not to rewrite, but to set up a process that structurally ensures that mistakes are either prevented or corrected. For a proof assistant, there are three main ways to do this: (1) *testing*, (2) *independent reimplementation*, and (3) *formal verification*. Lean has a fairly robust test suite, but testing helps more in the high-frequency low-impact domain; soundness bugs are rare and generally occur in areas that are insufficiently tested because the developer did not even consider the error condition.

The main contribution of this paper lies in (2): an independent reimplementation. To restore the soundness story of Lean to a satisfactory state, we believe it is necessary to bring back the external checkers. And what better language to do so than Lean itself? There are reasons not to use Lean for a truly *independent* verifier (and more external checkers are coming, in Scala[4] and Rust[5]), 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 Scala, Lean is designed for both programming and proving, so a kernel written in Lean can be formally verified. And we have good reasons for wanting the kernel to satisfy certain properties.

In this paper, we will present Lean4Lean,[6] 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), Batteries (the extended standard library), and Mathlib (the Lean mathematics library).[7] 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

[4] https://github.com/gebner/trepplein
[5] https://github.com/ammkrn/nanoda_lib
[6] https://github.com/digama0/lean4lean
[7] Lean4Lean can also check its own source code, although we did not benchmark this configuration specifically.

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 report partial progress, is the specification of Lean's metatheory and verification of the Lean4Lean kernel with respect to that theory, covering mistake prevention measure (3). This is similar to the MetaRocq project [28, 29], and amounts to a formalization of LTT [7]. Our principal contributions here are a definition of the typing judgment, some regularity theorems we can prove and others we can state, and the verification of some representative functions from the typechecker. The remainder is left as future work.

In this paper, we will avoid making value judgments about Lean or Rocq and their respective theories. These systems came to be what they are now as a result of complex historical factors, and we seek only to validate the behavior of Lean as it exists, even though it lacks some type theoretic properties that would have made the job easier if true. One possible outcome of this project is a recommendation for how to improve the typing rules, but this comes at a high cost to existing users so we do not take it lightly and we have no such recommendations currently.

Throughout the paper, code identifiers (both in listings and in inline prose) are hyperlinked to their definitions in the Lean4Lean and Lean repositories; we encourage readers to click through for additional context.

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 sets the stage for the verification of the typechecker, and how it was used to find a soundness bug in the kernel. Section 5 explains how the global structure of the environment is put together from individual typing judgments. Section 6 explains the complexities associated with supporting inductive types, treated as an extension of the base MLTT theory. Section 7 gives some performance results regarding the complete verifier, and Section 8 compares this project to MetaRocq, a Rocq formalization of Rocq metatheory and a verified kernel.

2 Base theory

2.1 Expressions

At the heart of the theory is the `VExpr`[8] type, which represents expressions of type theory. This is a minimalistic version of the type actually used by the kernel in section 3.1, to ease the burden of proving theoretical properties.[9]

```
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)
```

[8] 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.

[9] MetaRocq makes a similar but smaller separation: its expression type drops n-ary applications (replaced with binary) and casts (used to control evaluation strategies in Rocq), but otherwise mirrors the type used for metaprogramming. Our `VExpr`/`Expr` split goes further; see section 3.1 for the full list of differences.

- `bvar` n represents the nth 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.) Inductive constructors, recursors, and definitions are all represented using this constructor.
- `app` f a is function application, written $f\ a$ or $f(a)$.
- `lam` A e is a lambda-abstraction $\lambda x : A.\, e$. Because we are using de Bruijn variables, x is not represented; instead e is typechecked in an extended context where variable 0 has type A and the other variables are shifted up by 1.
- `forallE` A B [10] is a dependent Π-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 typechecking 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 LTT [7], 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 formalization uses `lift` and `subst` functions when moving expressions between contexts.)

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

2.2 The global environment

The typing relation is defined in a context of previously defined constants, called `VEnv`, with the definition:

```
structure VConstant where (uvars : Nat) (type : VExpr)
structure VDefEq where (uvars : Nat) (lhs rhs type : VExpr)
structure VEnv where
  constants : Name → Option (Option VConstant)
  defeqs : VDefEq → Prop
```

[10] The "`E`" suffix is used because `forall` is a keyword; we could use `forall` anyway but it would require escaping in several places.

This is a minimalistic environment type, essentially exactly what is needed to satisfy the requirements of the typing judgment. (The kernel's environment type is more complicated and will be discussed in Section 5.)

- `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.[11] This corresponds to the $\bar{u}.(c_{\bar{u}} : \alpha)$ declaration appearing in T-CONST below.
- `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 below.

2.3 Typing and definitional equality

For the typing rules, we use a slight variation of the typing judgments in LTT [7]; see Figure 1. Some remarks:

- LTT [7] uses two mutually inductive judgments $\Gamma \vdash e : \alpha$ and $\Gamma \vdash e_1 \equiv e_2$ (due to T-DEFEQ 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 section 2.4). Moreover Lean does not have good support for mutual inductive predicates, so keeping it single-inductive makes induction proofs easier.[12]
- We also define $\Gamma \vdash \alpha$ `type` to mean $\exists \ell.\ (\Gamma \vdash \alpha : \mathsf{U}_\ell)$ (incl. propositions $\Gamma \vdash p : \mathsf{U}_0$).
- 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. We should be upfront that the specific constraints on T-EXTRA required to handle inductive declarations are not yet pinned down: the typechecker accepts inductives in practice (see Section 6), but our metatheory does not yet close the loop on what extensions a valid inductive specification introduces. This is a known gap, and the largest remaining piece of future work on the theoretical side.

[11] 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.

[12] One could also encode the mutual structure as a single indexed inductive, but that doesn't substantially reduce the complexity – the intricacy here is intrinsic to the typing/def.eq. interaction, not the encoding.

$\boxed{\Gamma \ni x : \alpha}$

L-ZERO
$$\Gamma, x : \alpha \ni x : \alpha$$

L-SUCC
$$\frac{\Gamma \ni x : \alpha}{\Gamma, x' : \alpha' \ni x : \alpha}$$

$\boxed{\vdash \Gamma \ \mathsf{ok}}$

C-EMP
$$\vdash \cdot \ \mathsf{ok}$$

C-CONS
$$\frac{\vdash \Gamma \ \mathsf{ok} \quad \Gamma \vdash \alpha : \mathsf{U}_\ell}{\vdash \Gamma, x : \alpha \ \mathsf{ok}}$$

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

T-BVAR
$$\frac{\Gamma \ni x : \alpha}{\Gamma \vdash x : \alpha}$$

T-SYMM
$$\frac{\Gamma \vdash e \equiv e' : \alpha}{\Gamma \vdash e' \equiv e : \alpha}$$

T-TRANS
$$\frac{\Gamma \vdash e_1 \equiv e_2 : \alpha \quad \Gamma \vdash e_2 \equiv e_3 : \alpha}{\Gamma \vdash e_1 \equiv e_3 : \alpha}$$

T-SORT
$$\frac{n \vdash \ell, \ell' \ \mathsf{ok} \quad \ell \equiv \ell'}{\Gamma \vdash \mathsf{U}_\ell \equiv \mathsf{U}_{\ell'} : \mathsf{U}_{\mathrm{S}\,\ell}}$$

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

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

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

T-APP
$$\frac{\Gamma \vdash e_1 \equiv e_1' : \forall x : \alpha.\ \beta \quad \Gamma \vdash e_2 \equiv e_2' : \alpha}{\Gamma \vdash e_1\ e_2 \equiv e_1'\ e_2' : \beta[x \mapsto e_2]}$$

T-DEFEQ
$$\frac{\Gamma \vdash \alpha \equiv \beta : \mathsf{U}_\ell \quad \Gamma \vdash e \equiv e' : \alpha}{\Gamma \vdash e \equiv e' : \beta}$$

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

T-ETA
$$\frac{\Gamma \vdash e : \forall y : \alpha.\ \beta}{\Gamma \vdash (\lambda x : \alpha.\ e\ x) \equiv e : \forall y : \alpha.\ \beta}$$

T-PROOF-IRREL
$$\frac{\Gamma \vdash p : \mathsf{U}_0 \quad \Gamma \vdash h : p \quad \Gamma \vdash h' : p}{\Gamma \vdash h \equiv h' : p}$$

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.

2.4 Properties of the typing judgment

▶ **Lemma 1** (basic properties).

1. *(`IsDefEq.hasType`) If* $\Gamma \vdash e_1 \equiv e_2 : \alpha$*, then* $\Gamma \vdash e_1 : \alpha$ *and* $\Gamma \vdash e_2 : \alpha$*.*
2. *(`IsDefEq.closedN`) If* $\Gamma \vdash e \equiv e' : \alpha$ *and* Γ *is well-scoped, then* e*,* e'*, and* α *are well-scoped*[13] *(all free variables have indices less than* $|\Gamma|$*).*
3. *(`IsDefEq.weakN`, weakening) If* $\Gamma, \Gamma' \vdash e \equiv e' : \alpha$*, then* $\Gamma, \Delta, \Gamma' \vdash e \equiv e' : \alpha$*.*

[13] The formalization calls this property `ClosedN`, generalizing the standard sense of "closed" (no free variables) to a context of size N; the strict sense is recovered when $N = 0$.

4. *(`IsDefEq.instL`) If* $\Gamma \vdash_{E,n} e \equiv e' : \alpha$ *and* $\forall i.\ n' \vdash \ell_i$ ok*, then* $\Gamma[\bar{u} \mapsto \bar{\ell}] \vdash_{E,n'} e[\bar{u} \mapsto \bar{\ell}] \equiv e'[\bar{u} \mapsto \bar{\ell}] : \alpha[\bar{u} \mapsto \bar{\ell}]$.
5. *(`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. Immediate from T-SYMM and T-TRANS, recalling that $\Gamma \vdash e : \alpha$ means $\Gamma \vdash e \equiv e : \alpha$.
2. 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 5) 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.
3. By induction. We use Lemma 1.2 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 well-scoped.
4. By induction.
5. By induction on the first hypothesis. This uses Lemma 1.3 to lift $\Gamma \vdash e_0 : \beta$ 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-DEFEQ and weakening, and $\Gamma, x' : \alpha', x : \alpha \vdash e_1 \equiv e_2 : \beta$ by weakening, so $\Gamma, x : \alpha' \vdash e_1[x \mapsto x] \equiv e_2[x \mapsto x] : \beta[x \mapsto x]$ by Lemma 1.5. ◀

▶ **Lemma 3** (`HasType.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 $\Gamma \vdash e_1 \equiv e_2 : \gamma$ (recall $\Gamma \vdash e : \gamma$ means $\Gamma \vdash e \equiv e : \gamma$), assuming that one of e_1 or e_2 is $\forall x : \alpha.\ \beta$. We make repeated use of Lemma 1.1 to extract typing premises from def.eq. hypotheses. 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 Lemma 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 Lemma 1.5 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** (`HasType.sort_inv`). *If* $\Gamma \vdash_{E,n} \mathsf{U}_\ell : \gamma$ *then* $n \vdash \ell$ ok *(and therefore also* $\Gamma \vdash \mathsf{U}_\ell : \mathsf{U}_{\mathrm{S}\,\ell}$*).*

Proof sketch. Similar to Lemma 3. ◀

▶ **Theorem 5** (`IsDefEq.isType`). *If* $\vdash \Gamma$ ok *and* $\Gamma \vdash e : \alpha$ *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 Lemma 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 $\Gamma, x : \alpha \vdash \mathsf{U}_{\ell_2}$ type from the IH, so by Lemma 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* $\Gamma \vdash a \equiv a' : \alpha$ *then* $\Gamma \vdash f[x \mapsto a] \equiv f'[x \mapsto a'] : \beta[x \mapsto a]$.

Proof sketch. This is a generalization of Lemma 1.5, but we need Theorem 5 to prove that α and β are well-typed to apply the rules below.

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 because 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 $\mathsf{U}_\ell[x \mapsto a] \equiv \mathsf{U}_\ell[x \mapsto a'] : \mathsf{U}_{\mathrm{S}\,\ell}$, which is true by reflexivity. ◀

2.5 Conjectured properties of the typing judgment

Unfortunately, there are more structural properties of the typing judgment we need 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** (`IsDefEq.uniq`). *If* $\Gamma \vdash e : \alpha$ *and* $\Gamma \vdash e : \beta$, *then* $\Gamma \vdash \alpha \equiv \beta : \mathsf{U}_\ell$ *for some* ℓ.

Unique typing 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 in LTT [7].)*
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-DEFEQ *but with an untyped definitional equality.)*

This group of theorems is proved mutually with Conjecture 7 in LTT [7]:

▶ **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* $\Gamma, x : \alpha \vdash \beta \equiv \beta'$.[14]
3. *(`IsDefEqU.sort_forallE_inv`)* $\Gamma \vdash \mathsf{U}_\ell \not\equiv (\forall x : \alpha.\ \beta)$.

[14] This theorem is sometimes called "injectivity of ∀" but this name is slightly misleading because ∀ is not injective as a function on types, which is to say if we use propositional equality instead of definitional equality then the theorem fails.

The reason Conjecture 7 and Conjecture 9 have been downgraded from theorems in LTT [7] 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 believe the conjectures are true, but the induction needs more work. Lennon-Bertrand [21] explores proof approaches that could apply here, based on similar work for MetaRocq. Another route, taken by Siles and Herbelin [26] for general PTSs, uses a heavily annotated syntax in which terms carry their types explicitly, making unique typing essentially syntactic; the difficulty then relocates to elaborating the kernel's unannotated syntax into the annotated form, which is the typing problem in disguise. Soundness is unaffected, since alternative routes to the model avoid unique typing (also in LTT [7]), but these conjectures are needed in some form to prove correctness of the typechecker (see Section 3).

Another class of conjectured theorems concerns the "invertibility" of weakening:[15]

▶ **Conjecture 10** (Strengthening). *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 strengthening).

1. *(`IsDefEq.skips`) 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. *(`IsDefEq.weakN_iff`) 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. *(`OnCtx.weakN_inv`) If $\vdash \Gamma, \Delta, \Gamma'$ ok and Γ' does not mention the variables in Δ then $\vdash \Gamma, \Gamma'$ ok.*

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) (type body : Expr) (info : BinderInfo)
  | forallE (binderName : Name) (type body : Expr) (info : 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)
```

[15] This is not a trivial theorem, and MetaRocq found a bug in Rocq here; see section 3.5 of [27].

Comparing this with `VExpr` reveals some differences:

- `mvar` n is for metavariables in the elaborator. They are banned in kernel terms.
- `fvar` n is a "free variable", which is identified by a name rather than an index into the context. Although these are also banned, the kernel creates `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 the verification of the kernel.
- The `Level` type (not shown) also differs from `VLevel` similarly.
- The `lam` and `forallE` constructors contain additional elaborator metadata.
- The `letE` construct is used for `let x := e1; e2` expressions. We handle these by simply expanding them to `e2`$[$`x` $\mapsto$ `e1`$]$.
- The `lit` constructor is for literals, `Literal := Nat` $\oplus$ `String`. This is one of the new Lean 4 kernel features: naturals are bignums, and functions like `Nat.mul` are overridden with native implementations rather than using the inductive structure of `Nat` (effectively, unary numerals). For modeling in `VExpr` we just perform this (exponential-size) unfolding into `Nat.zero` and `Nat.succ`; similarly `String` is modeled as `List Char`, though at runtime it is a UTF-8 byte string.
- The `mdata` d e constructor is equivalent to e for modeling purposes (this is used for associating metadata to expressions).
- `proj` c i e is a projection out of a `structure` or structure-like inductive type (see Section 6). This is the hardest constructor to desugar: although it equals an application of the recursor, we need the type of e to build the term, so the desugaring is type-aware and only well-defined up to definitional equality.

Note that `Expr` was not defined as part of this project; it is imported directly from the Lean elaborator. In fact the C++ kernel reaches it via FFI, so Lean4Lean and the C++ kernel share 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 $E; \overline{u}; \Delta \vdash e \triangleright e'$ is shorthand for `TrExpr` E $\overline{u}$ Δ e e', asserting that in local context Δ, with names $\overline{u}$ for the local universe parameters and E for the global environment (see Section 5), e' : `VExpr` is a translation of e : `Expr`. It also asserts that Δ is well-typed and e' is well-typed in it, so $\exists e'.\ E; \overline{u}; \Delta \vdash e \triangleright e'$ asserts that e is a well-typed `Expr` – our target specification for correctness of the typechecker.

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 is a regular variable, the only kind we had in Section 2, while `vlet` α v is used inside 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 to relate them to de Bruijn indices. The name generator uses a global counter, with the invariant that every `FVarId` in the state is below it.

It is an `Option` because when typing an *open* term we must handle variables not yet assigned `FVarId`s. Lean does not process such terms directly, but they appear as subterms of ones it does, so we need both cases for 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. Since the two implementations do the same things in the same order, most bugs or peculiarities we discovered while formalizing Lean4Lean are replicated in the C++ kernel, giving us hope that verifying Lean4Lean will also lend strong evidence for the correctness of the C++ code.

3.2.1 Untangling recursion via RecM

The kernel consists of a large mutually-recursive set of functions. Because this is difficult for Lean to handle, and it also causes issues for proving theorems about the functions, it is desirable to break the dependencies. It turns out that just 4 kernel functions are enough to cut the mutual recursion: if each is split into two nodes, one receiving all incoming edges and one sending all outgoing edges, the call graph becomes acyclic (except for self-loops corresponding to normal recursion), so the remaining functions can be defined in dependency order. Conveniently, all four are also important high-level entry points:

1. `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.)
2. `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.
3. `whnfCore` e `cheapRec` `cheapProj` has the same specification as `whnf` e, but it lacks some early exit paths and does not unfold definitions. `cheapRec` and `cheapProj` are flags affecting which kinds of terms are unfolded. The reason `whnf` and `whnfCore` show up separately in this list is because they are both used in other functions (i.e. some functions call `whnfCore` directly because e.g. the fast path isn't applicable).
4. `inferType` e `inferOnly` infers or typechecks a term. That is, if `inferType` e `false` returns α then $\Gamma \vdash e : \alpha$ holds, and if `inferType` e `true` returns α then $\Gamma \vdash e : \alpha$ holds assuming $\Gamma \vdash e : \alpha'$ for some α'. (The latter mode is also known as *retyping* in Rocq.)

We use this to define the `RecM` monad:

```
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
```

Here `ReaderT` is the reader monad transformer, which augments the underlying `TypeChecker.M` monad with an immutable, implicitly-threaded `Methods` argument that any computation can read. `TypeChecker.M` itself contains the actual state of the typechecker.

But this method only kicks the problem up one step: how do we construct a `Methods` if every function needs another `Methods`? Ideally we would prove the kernel terminates, since dependent type theory (DTT) is meant to be terminating. However:

- It is unlikely that we can prove termination of a typechecker for Lean in Lean, because although the soundness proof in LTT [7] does not depend on termination, MetaRocq's does [29], 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:

  ```
  /-! Coquand-Abel construction of nontermination in proofs -/
  def True' := ∀ p : Prop, p → p
  def om : True' := fun A a =>
    @cast (True' → True') A  -- cast : α = β → α → β (reduces on α = α)
      (propext ⟨fun _ => a, fun _ => id⟩)  -- propext : (p ↔ q) → p = q
      (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[16] the standard solution for defining partial functions in a language like Lean or Rocq: 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

[16] Alternatives like the Bove-Capretta method [6] (take a termination proof as an argument) don't fit here: the kernel API is fixed to match Lean's, and we want it to time out on extreme cases anyway.

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.[17]

3.2.2 Inferring types

The kernel's typechecking algorithm is fairly straightforward. It is not bidirectional; instead it propagates the type of expressions forward via the `inferType` function, whose `inferOnly` flag is true when we may assume the expression is well-typed. For example, here is how $\lambda x : \alpha.\ e$ expressions are typed:

```
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
    if !inferOnly then
      let sort ← inferType d inferOnly
      _ ← ensureSortCore sort d
    withLocalDecl name d bi fun fv => do
      loop (fvars.push fv) body
  | e => do
    let e' := e.instantiateRev fvars
    let r ← inferType e' inferOnly
    let r := r.cheapBetaReduce
    return (← getLCtx).mkForall fvars r
```

Observe that this does more than simply checking T-LAM:

- Since it is using locally-nameless representation, whenever it needs to enter a binder it has to create a fresh `fvar` and replace `bvar 0`. The `withLocalDecl` function does this, and changes the local context to include the new `fvar`, but instantiation is deferred here.
- It actually consumes a *sequence* of lambdas, using the tail-recursive function `loop`. This is a performance optimization, since instantiation is $O(n)$ and so repeatedly instantiating the body in each step of the loop would cause $O(n^2)$ work. The actual instantiation is the two calls to `instantiateRev`.
- When `inferOnly` is true, meaning that it can assume the term is well-formed, it does not need to typecheck the type of the lambda.
- At the end of the loop, it uses `cheapBetaReduce` to reduce the result type in the common case that it has the form $(\lambda \overline{x}.\ x_i)\ \overline{e}$, or $(\lambda \overline{x}.\ c)\ \overline{e}$ (where c does not depend on $\overline{x}$).

We will discuss this function further in Section 4.

3.2.3 WHNF and reduction

The `whnf` function performs reductions at the "head" of an expression until a constructor is exposed. In so doing it implements a lazy, call-by-name evaluation strategy.

- A variable, sort, axiom, lambda, forall, or literal is in WHNF.
- On an application $f\ a$, reduce f to WHNF first. If it reduces to $\lambda x.\ e$, then β-reduce to $e[a/x]$ and continue.

[17] 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.

- On a let-expression $\mathsf{let}\ x := a;\ e$, reduce to $e[a/x]$ and continue.
- On the recursor of an inductive type, reduce its major argument first. If it reduces to a constructor, then β/ι-reduce (e.g. $\mathsf{rec}_{\mathbb{N}}\ C\ z\ s\ (\mathsf{suc}\ n) \rightsquigarrow s\ n\ (\mathsf{rec}_{\mathbb{N}}\ C\ z\ s\ n)$) and continue.
- On a definition, reduce to its body and continue.

Beyond these basics, there are a few additional Lean specifics:

- There is a list of 15 built-in functions on $\mathbb{N}$ like $+$, $-$, $\times$, div, mod, bitAnd, etc., which are evaluated in call-by-value order. For example:
  - If we are reducing $a + b$ to WHNF, then first reduce a and b. If these reduce to literals $[n]$, $[m]$, then compute $[n + m]$ and stop. (Compare: the default behavior on $a + 1$ would reduce to $\mathsf{suc}\ (a + 0)$ without reducing a.)
- When encountering $\mathsf{reduceBool}\ c$ where $c : \mathsf{Bool}$ is a constant definition, c is compiled to interpreter bytecode and executed. The result (`true` or `false`) is reified as an expression of type Bool as the result.

Supporting built-in functions on $\mathbb{N}$ in Lean4Lean was not difficult, but doing so soundly required implementing checks to ensure that e.g. when a definition named `Nat.add` is added to the environment, it satisfies the definitional equalities $a+0 \equiv a$ and $a+\mathsf{suc}\ n \equiv \mathsf{suc}\ (a+n)$. This allows us to prove by induction that this definition will compute on literals such that $[n] + [m] \equiv [n + m]$.[18]

On the other hand, supporting native evaluation using reduceBool is not nearly so easy. In theory, with a formalization of the IR (intermediate representation) semantics, it may be possible to prove the soundness of IR compilation and interpretation. However, IR can also call external functions from C, and with `implemented_by` one can completely decouple the compiler behavior from the kernel's version of the implementation, so the general feature is simply "unsound by design." Given that Mathlib and many other mathematics projects avoid reduceBool completely for these reasons, we do not lose much by not supporting it.

3.2.4 Definitional equality checking

This is the least "canonical" part of a DTT typechecker. The worst-case complexity is galactically large, so the heuristics are critical for keeping typechecking in the same ballpark as the original check by the human author (we only aim to check Lean projects in the wild, not adversarial inputs). The algorithm for checking $s \stackrel{?}{\equiv} t$ in Lean goes roughly as follows:

- If s and t are pointer-equal, or already known equal via a union-find cache of def.eq. subterms, return true.
- If s and t are both lambdas, sorts, foralls, or literals, then compare subterms.
- Validate $\mathsf{true} \equiv t$ if $t \rightsquigarrow \mathsf{true}$.[19]
- Reduce $s \rightsquigarrow s'$ and $t \rightsquigarrow t'$ with `cheapProj := true`.
- If $s' : \alpha$ and $t' : \beta$ and $\alpha \equiv \beta : \mathsf{U}_0$ then return true (T-PROOF-IRREL).
- If $s' = f\ \overline{a}$ and $t' = g\ \overline{b}$ where f, g are definitions: if $f = g$ and $a_i \equiv b_i$ for all i, return true; otherwise unfold the side with lower height[20] and continue.
- If $s' = t'$ are variables, or $s' = c_{\bar{\ell}}$ and $t' = c_{\bar{\ell}'}$ with $\bar{\ell} \equiv \bar{\ell}'$, or $s' = u.i$ and $t' = v.i$ with $u \equiv v$, return true.

[18] The original Lean kernel does not perform these checks. This is technically unsound, but to exploit it one would need to replace the prelude, and this is not a supported configuration.
[19] This is an optimization for proof by reflection, where a goal $t = \mathsf{true}$ is proved using $\mathsf{refl}\ \mathsf{true} : \mathsf{true} = \mathsf{true}$.
[20] $\mathsf{height}(f) = 1 + \max_{c \preceq \delta(f)} \mathsf{height}(c)$, where $\delta(f)$ is the body of definition f.

- Reduce $s' \rightsquigarrow s''$ and $t' \rightsquigarrow t''$ again, this time with `cheapProj := false` (which can fire projection reductions skipped in the first pass).
- If $s'' = f\ \overline{a}$ and $t'' = g\ \overline{b}$ and $f \equiv g$ and $a_i \equiv b_i$ for all i, return true.
- Try the η-laws (in each case possibly swapping s and t): η-expansion ($t'' = \lambda \overline{x}.\ u$ with $s''\ \overline{x} \equiv u$), structure η ($t'' = (\overline{u_i})$ with $s''.i \equiv u_i$ for all i), and unit η ($s'' : \mathsf{Unit}$ or another unit-like structure); also unfold string literals.
- Otherwise, fail. (Because of various incompletenesses in this algorithm, this doesn't actually imply $s \not\equiv t$.)

4 Verifying the algorithm

We would now like to connect the results of Section 2 and Section 3 by showing that the monadic typechecker functions satisfy their respective specifications. The typechecker monad `M` is a standard reader/state/exception monad stack:

```
abbrev M := ReaderT Context <| StateT State <| Except Exception
```

Because we give the semantics of `Expr` using `VExpr`, and this translation is only unique up to def.eq. (definitional equality), we will maintain a shadow state `VContext` and `VState` which have `Expr` × `VExpr` pairs everywhere so that we can project out both the original `Context` as well as the corresponding `VExpr` for each well-formed expression.

We use a lightweight monadic program logic based on Hoare logic, but "one-sided": for the precondition, we pass the exact context and state (and additional hypotheses become regular preconditions on the theorem), and for the postcondition we have a predicate over possible next state and return values.

```
def M.WF (c : VContext) (vs : VState) (x : M α) (Q : α → VState → Prop) :=
  vs.WF c → ∀ a s', x c.toContext vs.toState = .ok (a, s') →
    ∃ vs', vs'.toState = s' ∧ vs ≤ vs' ∧ vs'.WF c ∧ Q a vs'
```

Here c : `VContext` and vs : `VState` are the shadow context and state, and we write $\widehat{c} := c$.`toContext` and $\widehat{vs} := vs$.`toState` for their projections to the underlying `Context` and `State` actually threaded through the `M` monad. In words, the definition says: assuming (c, vs) are well-formed, if running the monadic computation x : `M` α on $(\widehat{c}, \widehat{vs})$ evaluates to some $a : \alpha$ in state s', then there exists vs' with $\widehat{vs'} = s'$ such that $vs \le vs'$ (a partial order on states, which intuitively says that vs' extends vs forward in time), (c, vs') is well-formed, and $Q(a, vs')$ holds. The $\le$ relation is used mainly to handle the global counter for creating fresh `fvars`: the well-formedness invariant ensures the current counter value is greater than any `fvar` in the state or context, and $vs \le vs'$ enforces that the counter only ever grows, keeping the state and context well-formed.

We can use this to define the specification of the `Methods`:

```
structure Methods.WF (m : Methods) where
  isDefEqCore : c.TrExpr e₁ e₁' → c.TrExpr e₂ e₂' →
    (m.isDefEqCore e₁ e₂).WF c s fun b _ => b → c.IsDefEqU e₁' e₂'
  whnfCore : c.TrExpr e e' →
    (m.whnfCore e cr cp).WF c s fun e₁ _ => c.FVarsBelow e e₁ ∧ c.TrExpr e₁ e'
  whnf : c.TrExpr e e' →
    (m.whnf e).WF c s fun e₁ _ => c.FVarsBelow e e₁ ∧ c.TrExpr e₁ e'
  inferType : e.FVarsIn (· ∈ c.vlctx.fvars) →
    (inferOnly = true → ∃ e', c.TrExpr e e') →
    (m.inferType e inferOnly).WF c s fun ty _ => ∃ e' ty', c.TrTyping e ty e' ty'
```

- `isDefEqCore` (and `isDefEq`) ensures that if the inputs are well-typed expressions $e_1 \triangleright e_1'$ and $e_2 \triangleright e_2'$, and it returns `true`, then $e_1' \equiv e_2'$.
- `whnf` and `whnfCore` ensure that given a well-formed expression e, if they return e_1 then $e \equiv e_1$. We can succinctly say this by assuming $e \triangleright e'$ and asserting $e_1 \triangleright e'$ as well, since $\triangleright$ is closed under def.eq. (There is also a technical condition $\mathrm{FV}(e) \supseteq \mathrm{FV}(e_1)$.)
- `inferType` does not assume that it has a well-formed input, but it does need to know that the global counter is fresh for any fvars in the input. If `inferOnly` is true, it also assumes e is well-formed. It asserts that $e \triangleright e'$ and $\tau \triangleright \tau'$ are well-formed, and $e' : \tau'$ – the `VExpr` witnesses are needed because typing is defined at the `VExpr` level, and the same trick as for `whnf` above gives Expr-side well-typedness of e and τ from a single statement.

We then define `RecM.WF` with a similar signature to `M.WF` to mean that it is valid when called with a well-formed `Methods`.

The specification of `inferLambda` from section 3.2.2 should now be unsurprising:

```
theorem inferLambda.WF
  (h1 : e.FVarsIn s.ngen.Reserves)
  (hinf : inferOnly = true → ∃ e', c.TrExpr e e') :
  (inferLambda e inferOnly).WF c s fun ty _ =>
    ∃ e' ty', c.TrExpr e e' ∧ c.TrExpr ty ty' ∧ c.HasType e' ty'
```

The theorem is proved compositionally over the code. At the loop, we need an inductive invariant which reveals some of the complications of having a sequence of fvars that have not yet been applied to some of the inputs:

```
theorem inferLambda.loop.WF
  {c : VContext} {e₀ : Expr}
  {m} [mwf : c.MLCWF m] {n} (hn : n ≤ m.length)
  (hdrop : m.dropN n hn = c.mlctx)
  (harr : arr.toList.reverse = (m.fvarRevList n hn).map .fvar)
  (he₀ : e₀ = m.mkLambda n hn ei)
  (hei : e.instantiateList ((m.fvarRevList n hn).map .fvar) = ei)
  (hbelow : ∀ P, IsFVarUpSet P c.vlctx → FVarsIn P e₀ →
    IsFVarUpSet (AllAbove c.vlctx P) m.vlctx ∧ FVarsIn (AllAbove c.vlctx P) ei ∧
    ∀ ty, FVarsIn (AllAbove c.vlctx P) ty →
      FVarsIn (AllAbove c.vlctx P) (m.mkForall n hn ty))
  (hr : e.FVarsIn (· ∈ m.vlctx.fvars))
  (hinf : inferOnly = true → ∃ e', (c.withMLC m).TrExprS ei e') :
  (inferLambda.loop inferOnly arr e).WF (c.withMLC m) s fun ty _ =>
    ∃ e' ty', c.TrTyping e₀ ty e' ty'
```

Here m : `MLCtx` is the aforementioned local context consisting of `Expr` × `VExpr` pairs, which we use to locally extend the local context while leaving the rest of the `VContext` alone. Most of the hypotheses are gluing equations: m extends `c.mlctx` by n binders, `arr` holds the fresh fvars for them, e_i is e with those bvars instantiated, and e_0 is the reconstructed input lambda. The more interesting machinery is `hbelow`, which is a good entry point into the verification toolkit we use throughout:

- An `IsFVarUpSet` predicate P over a context Δ is a set of fvars closed under the dependency graph of Δ: if $v \in P$ and v has a type or value (in Δ) containing v', then $v' \in P$ as well.
- `AllAbove` Δ P constructs an fvar set on a larger context Δ' as $P \cup (\Delta' \setminus \Delta)$, including all extension variables $x \in \Delta' \setminus \Delta$. Asserting this is an up-set therefore asserts that all extension variables have types definable from variables in P.

With this, `hbelow` says: any up-set property of e_0 over `c.vlctx` lifts (via `AllAbove`) to one over the extended `m.vlctx` that is also preserved by e_i and by wrapping a result type in `m.mkForall`. The intuition is cleanest in the strongest case, where P is the fvar set of e_0 itself: the loop is decomposing e_0 into e_i and a stack of binders, so both e_i and the binder types in m are subterms of e_0 and trivially share its fvars. The general $\forall\, P$ form makes the proofs more compositional. The remaining `mwf`, `hr`, and `hinf` mirror the preconditions of `inferLambda.WF`, lifted to the extended context. Crucially, the conclusion gives a type for e_0, the original input, at the original context c, so it works with the tail recursive structure of the function.

4.1 Verification status

The main body of the typechecker has been fully verified along the lines above. That is, the four functions from section 3.2.1 are proved to satisfy the specifications from Section 4, and so by induction on recursion depth the functions all satisfy their specifications when lifted out of the `RecM` monad as well, with theorems such as `checkType.WF`. The areas that are still unfinished include:

- Anything related to inductive types (Section 6) or primitive projections;
- Unique typing and injectivity of type constructors (section 2.5);
- Verification of an experimental complete level equality algorithm following [15];
- Environment modifications (Section 5).

Of these, the most significant is the last one, since the kernel cannot be used without the `addDecl` function. But this at least shows that the kernel can act as a typechecker given a fixed environment.

4.2 Bit tricks and soundness bugs

However, in the course of performing the verification we have already run into some interesting issues, and at least one soundness bug (which has since been fixed in response to our report).

Recall that `inferLambda` calls `cheapBetaReduce`, so we need to prove that `cheapBetaReduce` (which morally performs only β-reduction) preserves def.eq. The `cheapBetaReduce` function in turn uses `hasLooseBVars` to determine whether the function is a constant function (since $\lambda \overline{x}.\ c$ is a constant function if c has no bvars after stripping all the binders without instantiating them). `hasLooseBVars` e is defined as `looseBVarRange` e `> 0`, so we are reduced to verifying `looseBVarRange`, which returns the largest unbound bvar.

One would expect this to be defined recursively over `Expr`, with $\mathsf{LBR}(\mathsf{bvar}\ i) = i + 1$, $\mathsf{LBR}(\mathsf{lam}\ \tau\ e) = \max(\mathsf{LBR}(\tau), \mathsf{LBR}(e) - 1)$, and so on. However, it is instead implemented by masking off 20 bits from a 64-bit data field $e.\mathsf{data}$ which is stored with every expression. So this naturally leads to the question: what happens on overflow? The calculation of $e.\mathsf{data}$ had an overflow check, but it was expressed in Lean using the `panic!` function, which signals an error on `stdout` but then continues execution with the default value of the type, which is 0 for `UInt64`, which also happens to be the worst possible answer in this case, since it effectively turns on all of the optimizations, including the one in question about `hasLooseBVars`. Concretely (using a variant of the bug for level expressions), $f^{2^{24}}(u)$ where $f(\ell) = \max(0, \ell)$ is a level expression which normalizes to u, but because the u is so deeply nested in the term it triggers the overflow and so Lean believes it to not have a variable in it, from which one can derive $f^{2^{24}}(u) \equiv f^{2^{24}}(v)$ and thus $\mathsf{U}_0 \equiv \mathsf{U}_1$.

This is a nice example of a bug that was found entirely theoretically, by failing to prove a theorem and working backwards to an actual exploitable soundness bug.

Unfortunately, the fix does not address all of the problems we identified. The way it was addressed was to make the `Expr.data` function opaque, which at most masks the issue; and also by moving the panic to C++ code, where it can "panic harder" and crash the program, so that the strange consequences of returning 0 are not observed. Using opaques to replace a low level definition with a model is a reasonable option, but currently we are still relying on an unsound assumption (`looseBVarRange_eq`), because the definition of `looseBVarRange` is still visible and so one can prove that e.g. `looseBVarRange` e `< 2^20` unconditionally, which means that `looseBVarRange` cannot be the "correct" definition, which means the theorem isn't provable even though all counterexamples are unreachable (but Lean's logic doesn't know that).

5 Adding to the global environment

As with `VExpr` vs `Expr`, the kernel does not directly use the type `VEnv` from section 2.2; it uses `Environment` instead to store declarations. This type is complex and contains many details irrelevant to the kernel, but luckily we only really care about a few operations on it:

- `empty` : `Environment` – constructs an environment with no constants
- `add` : `Environment` → `ConstantInfo` → `Environment` – adds a `ConstantInfo` declaration to the environment
- `find?` : `Environment` → `Name` → `Option ConstantInfo` – retrieves a declaration by name

In effect, the environment is a fancy hashmap indexing `ConstantInfo` declarations by name. Not all extensions of a `VEnv` are legal: as defined there is no constraint on definitional equalities, but Lean only allows them in a few cases – definition unfolding, recursor reduction, and quotient reduction – captured in `Environment` through `ConstantInfo` additions (definitions, theorems, and so on).

So to express environment well-formedness we require it be built up recursively from nothing by adding any kind of well-typed declaration `VDecl`. The declarations are:

- `block` n blocks constant n, setting `env.constants` n `= some none` and preventing a re-declaration as mentioned above.
- `axiom { name :=` c`, uvars :=` n`, type :=` α `}` adds $c : \alpha$ as an axiom to the constant map.
- `opaque { `c`, `n`, `α`, value :=` e `}` checks that $e : \alpha$, then adds $c : \alpha$ to the constant map.
- `def { `c`, `n`, `α`, `e `}` does the same as `opaque` but also adds $c \equiv e : \alpha$ to `defeqs`.
- `example { uvars :=` n`, type :=` α`, value :=` e `}` checks that $e : \alpha$ and leaves the environment as-is.
- `induct (`d `: VInductDecl)` adds all the constants and def.eq.'s from an inductive declaration (see Section 6).
- `quot` adds the quotient axioms and definitional equality. This is a type operator α/R where $R : \alpha \to \alpha \to$ `Prop` with mk : $\alpha \to \alpha/R$ and 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 (lift $f\ h$ (mk a)) $\equiv f\ a : \beta$). (One can construct every part of this in Lean except for the definitional equality.)

This enumeration is similar to the `Declaration` type, the actual front end to the kernel, but it lacks `unsafe` declarations and `mutualDefnDecl` (which, despite the name, is not for mutual definitions but rather unsafe declarations with unchecked self-referential definitions). We deliberately do not model `unsafe` declarations: as certain checks are disabled, they no longer follow the theory, and in any case they play no role in checking theorems.

6 Inductive types

Lean's implementation of inductive types is based on Dybjer [13]. It differs from Rocq in that rather than primitive fix and match expressions, we have constants called "recursors" generated from the inductive specification. This significantly simplifies `VExpr`, which has no special support for anything inductive-related beyond the generic def.eq. hook T-EXTRA, and means we need no complex guard checker – something MetaRocq axiomatizes [29] and a past source of soundness bugs. Instead the complexity moves to recursor generation.

For single inductives, the algorithm is described in full in LTT [7]. In Lean 3, nested and mutual inductives were simulated using single inductives so the kernel only had to deal with the single inductive case, but for performance reasons Lean 4 moved the checking to the kernel. A mutual block such as `Even` and `Odd` over `Nat`, with `Even.succ : Odd n → Even (n+1)` and `Odd.succ' : Even n → Odd (n+1)` generates the expected constructors together with one recursor per type, `Even.rec` and `Odd.rec`. Each recursor takes a motive and minor premises for *both* types in the block, so that eliminating an `Even n` can recurse into its nested `Odd` subterms. The associated ι-reduction definitional equalities fire when a recursor meets a constructor, dispatching to the matching minor premise and recursing through the sibling recursor; e.g. `@Even.rec .. (Even.succ e)` reduces to `@Fs n e (@Odd.rec .. 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.

Currently, Lean4Lean contains a complete implementation of inductive types, including nested and mutual inductives and all the typechecking effects of this, but not much work has been done on the theoretical side, defining what an inductive specification should generate.

6.1 Eta for structures

Another new kernel feature in Lean 4 is primitive projections and η for structures: if s is a `structure` (a one-constructor non-recursive `inductive` type), then $s \equiv \mathsf{mk}(s.1, \ldots, s.n)$ where $s.i$ is the ith projection (`Expr.proj`). This already held propositionally (by induction on s), but is now definitional, so the kernel must unfold structures as needed. η for unit in particular is known to be problematic [17, 19], but Lean's kernel does not implement a complete decision procedure anyway; we believe it poses no soundness issue and the main results should still hold even with this extension.

7 Results

Lean4Lean contains a command-line frontend for validating already-compiled Lean projects. It reads `.olean` files, essentially serialized (unordered) lists of `ConstantInfo` objects, and topologically sorts the definitions and passes them to the Lean4Lean kernel, a function

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

that is a drop-in replacement for `Environment.addDeclCore`, 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`,[21] which does the same but calls Lean's `addDecl` instead, using the C++ kernel as an external verifier. Using the Lean kernel as an external verifier for itself may seem odd, but it catches malicious

[21] `https://github.com/leanprover/lean4checker`

	`lean4checker`	`lean4lean`	ratio
Lean	37.01 s	44.61 s	1.21
Batteries	32.49 s	45.74 s	1.40
Mathlib + Batteries + Lean	44.54 min	58.79 min	1.32

Figure 2 Comparison of the original C++ implementation of the Lean kernel (`lean4checker`) 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`.

(or confused) code that uses metaprogramming to bypass the kernel and add constants to the environment without typechecking. For our purposes it is a handy benchmark, since we perform the same operations with a different kernel. Running `lake env lean4lean --fresh Mathlib` in `mathlib4` will check the Mathlib module and all of its dependencies.[22]

The results are summarized in Figure 2. The new implementation is around 30% slower than the C++ implementation, which we attribute mainly to the Lean compiler and data representation; nevertheless it is practically usable on large libraries, a strong sign. We do not expect it to get faster without compiler improvements, but we intend to keep the speed from regressing as verification grows; the external verification makes this possible in principle, though more tooling may be needed to verify the implementation as-is in practice.

8 Related work

Self-verification of ITP systems has been done in Rocq [4], HOL Light [16], Isabelle [22], and Metamath Zero [8], and also overlaps with bootstrapping theorem provers such as Milawa [11] and Candle [2, 18]. But the most similar work is unquestionably MetaRocq [28, 29, 30], which is a project to develop a verified typechecker for Rocq, in Rocq. It is significantly more complete than the previous effort [4], but is not yet capable of verifying the Rocq standard library due to lack of eta-conversion and template polymorphism; eta-conversion is work in progress [19], and universe polymorphism being reorganised in Rocq's kernel [24]. 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 feature-complete typechecker but we are still lacking some theorems. Our work also overlaps to some extent with Template-Rocq [3, 5, 33], a certified metaprogramming framework for Rocq developed in the MetaRocq repo. A parallel effort for Agda, Agda Core [9, 10], is currently in development.

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

- In MetaRocq, the conversion relation is a partial order rather than an equivalence relation, because of universe cumulativity, so Conjecture 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 expressed in a different way in Rocq.
- Lean's typechecker is known to not be complete, owing to some of the undecidability results in LTT [7], but MetaRocq [30, 20] has a proof of completeness.

[22] Single-threaded; per-module parallel checking is possible but harder to benchmark.

9 Conclusion

We have presented Lean4Lean, a reimplementation of the Lean 4 typechecker written in Lean itself and closely following the kernel. Despite its simplicity, Lean4Lean can validate large-scale developments such as Mathlib with competitive performance.

More importantly, Lean4Lean serves as a basis for formal reasoning about Lean's metatheory. By internalizing the type system, we can verify properties of the checker and use these results to increase confidence in the production kernel. Already, this process has revealed subtle issues and motivated fixes upstream.

Looking ahead, we plan to formalize more of LTT [7], including in particular the proof of relative consistency given a large cardinal assumption. By continuing the verification effort, we hope to discover more potential bugs in the production kernel. In the long term, we hope that this work will lead to a fully verified, self-hosted Lean kernel.

References

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

2 Oskar Abrahamsson, Magnus O. Myreen, Ramana Kumar, and Thomas Sewell. Candle: A Verified Implementation of HOL Light. In June Andronick and Leonardo de Moura, editors, *13th International Conference on Interactive Theorem Proving, ITP 2022, Haifa, Israel, August 7-10, 2022*, volume 237 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 3:1–3:17, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ITP.2022.3.

3 Abhishek Anand, Simon Boulier, Cyril Cohen, Matthieu Sozeau, and Nicolas Tabareau. Towards Certified Meta-Programming with Typed Template-Coq. In Jeremy Avigad and Assia Mahboubi, editors, *Interactive Theorem Proving - 9th International Conference, ITP 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, volume 10895 of *Lecture Notes in Computer Science*, pages 20–39, Oxford, United Kingdom, July 2018. Springer. doi:10.1007/978-3-319-94821-8_2.

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

5 Simon Pierre Boulier. *Extending type theory with syntactic models. (Etendre la théorie des types à l'aide de modèles syntaxiques)*. Theses, Ecole nationale supérieure Mines-Télécom Atlantique Bretagne Pays de la Loire, France, November 2018. URL: https://tel.archives-ouvertes.fr/tel-02007839.

6 Ana Bove and Venanzio Capretta. Modelling general recursion in type theory. *Mathematical Structures in Computer Science*, 15(4):671–708, 2005. doi:10.1017/S0960129505004822.

7 Mario Carneiro. The Type Theory of Lean. Master's thesis, Carnegie Mellon University, 2019. URL: https://github.com/digama0/lean-type-theory/releases/tag/v1.0.

8 Mario Carneiro. Metamath Zero: Designing a Theorem Prover Prover. In Christoph Benzmüller and Bruce R. Miller, editors, *Intelligent Computer Mathematics - 13th International Conference, CICM 2020, Bertinoro, Italy, July 26-31, 2020, Proceedings*, volume 12236 of *Lecture Notes in Computer Science*, pages 71–88, Berlin, Heidelberg, 2020. Springer. doi:10.1007/978-3-030-53518-6_5.

9 Jesper Cockx. Agda Core. https://github.com/jespercockx/agda-core, 2024.

10 Jesper Cockx. Agda Core: The Dream and the Reality. Talk at IFIP WG 2.8, https://jesper.cx/files/IFIP28-2024.html, April 2024.

11 Jared Curran Davis. *A self-verifying theorem prover*. PhD thesis, University of Texas, 2009.

12 Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean Theorem Prover (System Description). In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25 - 25th International Conference on*

Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings, volume 9195 of *Lecture Notes in Computer Science*, pages 378–388. Springer, Springer, 2015. doi:10.1007/978-3-319-21401-6_26.

13 Peter Dybjer. Inductive Families. *Formal Aspects of Computing*, 6(4):440–465, 1994. doi:10.1007/BF01211308.

14 Gabriel Ebner. trepplein: a Lean 3 Type Checker. https://github.com/gebner/trepplein, 2022.

15 Yoan Géran. A Canonical Form for Universe Levels in Impredicative Type Theory. In Stefano Guerrini and Barbara König, editors, *34th EACSL Annual Conference on Computer Science Logic, CSL 2026, Paris, France, February 23-28, 2026*, volume 363 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 39:1–39:21. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2026. doi:10.4230/LIPIcs.CSL.2026.39.

16 John Harrison. Towards Self-verification of HOL Light. In Ulrich Furbach and Natarajan Shankar, editors, *Automated Reasoning, Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4130 of *Lecture Notes in Computer Science*, pages 177–191, Seattle, WA, 2006. Springer. doi:10.1007/11814771_17.

17 András Kovács. Eta conversion for the unit type (is still not that simple). In *WITS 2025 - Workshop on the Implementation of Type Systems*, 2025. URL: https://popl25.sigplan.org/details/wits-2025-papers/4/.

18 Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: a verified implementation of ML. In Suresh Jagannathan and Peter Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 179–192, New York, NY, USA, January 2014. ACM. doi:10.1145/2535838.2535841.

19 Meven Lennon-Bertrand. À bas l'η — Coq's troublesome η-conversion. In *Proceedings of the First Workshop on the Implementation of Type Systems (WITS)*, 2022. URL: https://www.meven.ac/documents/22-WITS-abstract.pdf.

20 Meven Lennon-Bertrand. *Bidirectional Typing for the Calculus of Inductive Constructions. (Typage Bidirectionnel pour le Calcul des Constructions Inductives)*. Theses, University of Nantes, France, June 2022. URL: https://tel.archives-ouvertes.fr/tel-03848595.

21 Meven Lennon-Bertrand. What Does It Take to Certify a Conversion Checker? In Maribel Fernández, editor, *10th International Conference on Formal Structures for Computation and Deduction, FSCD 2025, Birmingham, UK, July 14-20, 2025*, volume 337 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 27:1–27:23, Dagstuhl, Germany, 2025. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.FSCD.2025.27.

22 Tobias Nipkow and Simon Roßkopf. Isabelle's Metalogic: Formalization and Proof Checker. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings*, volume 12699 of *Lecture Notes in Computer Science*, pages 93–110, Berlin, Heidelberg, 2021. Springer. doi:10.1007/978-3-030-79876-5_6.

23 Christine Paulin-Mohring. Introduction to the Calculus of Inductive Constructions. In Bruno Woltzenlogel Paleo and David Delahaye, editors, *All about Proofs, Proofs for All*, volume 55 of *Studies in Logic (Mathematical logic and foundations)*. College Publications, January 2015. URL: https://inria.hal.science/hal-01094195.

24 Josselin Poiret, Gaëtan Gilbert, Kenji Maillard, Pierre-Marie Pédrot, Matthieu Sozeau, Nicolas Tabareau, and Éric Tanter. All Your Base Are Belong to Us: Sort Polymorphism for Proof Assistants. *Proceedings of the ACM on Programming Languages*, 9(POPL):2253–2281, January 2025. Conference talk, Session: POPL Research Papers, Fri 24 Jan 2025 17:40–18:00, Proof Assistants track. doi:10.1145/3704912.

25 Daniel Selsam. tc: Reference Type Checker for Lean 3. https://github.com/dselsam/tc, 2017.

26 Vincent Siles and Hugo Herbelin. Pure Type System conversion is always typable. *Journal of Functional Programming*, 22(2):153–180, 2012. doi:10.1017/S0956796812000044.

27 Matthieu Sozeau. Touring the MetaCoq Project (Invited Paper). In Elaine Pimentel and Enrico Tassi, editors, *Proceedings of the Sixteenth Workshop on Logical Frameworks and Meta-Languages: Theory and Practice, LFMTP 2021, Pittsburgh, USA, 16th July 2021*, volume 337 of *Electronic Proceedings in Theoretical Computer Science (EPTCS)*, pages 13–29, Pittsburgh, United States, July 2021. doi:10.4204/EPTCS.337.2.

28 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*, 64(5):947–999, February 2020. doi:10.1007/s10817-019-09540-0.

29 Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. Coq Coq correct! verification of type checking and erasure for Coq, in Coq. *Proceedings of the ACM on Programming Languages*, 4(POPL):8:1–8:28, December 2020. doi:10.1145/3371076.

30 Matthieu Sozeau, Yannick Forster, Meven Lennon-Bertrand, Jakob Botsch Nielsen, Nicolas Tabareau, and Théo Winterhalter. Correct and Complete Type Checking and Certified Erasure for Coq, in Coq. *Journal of the ACM*, 72(1):8:1–8:74, January 2025. doi:10.1145/3706056.

31 The Coq Development Team. The Coq Proof Assistant, April 2020. doi:10.5281/zenodo.3744225.

32 Sebastian Andreas Ullrich. *An Extensible Theorem Proving Frontend*. PhD thesis, Karlsruhe Institute of Technology, Germany, 2023. doi:10.5445/IR/1000161074.

33 Théo Winterhalter. *Formalisation and Meta-Theory of Type Theory. (Formalisation et Méta-Théorie de la Théorie des Types)*. PhD thesis, University of Nantes, France, 2020. URL: https://tel.archives-ouvertes.fr/tel-05425836.