A Certifying Proof Assistant for Synthetic Mathematics in Lean

Wojciech Nawrocki
Carnegie Mellon University
Pittsburgh, USA
wjnawrocki@cmu.edu

Joseph Hua
Carnegie Mellon University
Pittsburgh, USA
jlhua@andrew.cmu.edu

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

Yiming Xu
LMU Munich
Munich, Germany
yiming.xu@lmu.de

Spencer Woolfson
Chapman University
Orange, USA
swoolfso@andrew.cmu.edu

Shuge Rong
Carnegie Mellon University
Pittsburgh, USA
shuger@andrew.cmu.edu

Sina Hazratpour
Stockholm University
Stockholm, Sweden
sinahazratpour@gmail.com

Steve Awodey
Carnegie Mellon University
Pittsburgh, USA
awodey@cmu.edu

Abstract

Synthetic theories such as homotopy type theory axiomatize classical mathematical objects such as spaces up to homotopy. Although theorems in synthetic theories translate to theorems about the axiomatized structures on paper, this fact has not yet been exploited in proof assistants. This makes it challenging to formalize results in classical mathematics using synthetic methods. For example, Cubical Agda supports reasoning about cubical types, but cubical proofs have not been translated to proofs about cubical set models, let alone their topological realizations.

To bridge this gap, we present `SynthLean`: a proof assistant that combines reasoning using synthetic theories with reasoning about their models. `SynthLean` embeds Martin-Löf type theory as a domain-specific language in Lean, supporting a bidirectional workflow: constructions can be made internally in Martin-Löf type theory as well as externally in a model of the theory. A certifying normalization-by-evaluation typechecker automatically proves that internal definitions have sound interpretations in any model; conversely, semantic entities can be axiomatized in the syntax. Our implementation handles universes, Σ, Π, and identity types, as well as arbitrary axiomatized constants. To provide a familiar experience for Lean users, we reuse Lean's tactic language and syntax in the internal mode, and base our formalization of natural model semantics on Mathlib. By taking a generic approach, `SynthLean` can be used to mechanize various interpretations of internal languages such as the groupoid, cubical, or simplicial models of homotopy type theory in `HoTTLean`.

CCS Concepts: • **Theory of computation → Type theory**; • **Mathematics of computing → Mathematical software**.

Keywords: type theory, categorical logic, proof assistants, Lean

ACM Reference Format:
Wojciech Nawrocki, Joseph Hua, Mario Carneiro, Yiming Xu, Spencer Woolfson, Shuge Rong, Sina Hazratpour, and Steve Awodey. 2026. A Certifying Proof Assistant for Synthetic Mathematics in Lean. In *Proceedings of the 15th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '26), January 12–13, 2026, Rennes, France.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3779031.3779087

CPP '26, Rennes, France

ACM ISBN 979-8-4007-2341-4/2026/01
https://doi.org/10.1145/3779031.3779087

1 Introduction

Our work serves to expand the scope of formalization to proofs relying on *synthetic* methods. Consider the following proof of a classical result from homotopy theory.

Theorem 1.1. *The fundamental group $\pi_1(S^1)$ of the topological circle is $\mathbb{Z}$.*

Proof. (1) In homotopy type theory (HoTT), we define S^1 as the higher inductive type

```
inductive S¹ where
  | base : S¹
  | loop : base = base
```

with the loop space $\Omega(S^1, \mathrm{base})$ at the basepoint as the identity type base = base, and define the integers $\mathbb{Z}$ as an inductive type. A type-theoretic construction using the univalence axiom [52] proves $(\mathrm{base} = \mathrm{base}) = \mathbb{Z}$.

(2) Now note that in the groupoid model [41] of HoTT (restricted to 1-truncated types), the type base = base is interpreted as the group(oid) of automorphisms on the object $[\![\mathsf{base}]\!]$ in the groupoid $[\![S^1]\!]$, and $\mathbb{Z}$ is interpreted as the ordinary integers viewed as a discrete groupoid. By soundness of interpretation we have

$$\mathsf{Aut}_{[\![S^1]\!]}[\![\mathsf{base}]\!] \cong_{\mathbf{Grpd}} \mathbb{Z}$$

(3) The groupoid $[\![S^1]\!]$ can be realized topologically as a CW complex which is homeomorphic to the topological circle $\{(x, y) \in \mathbb{R}^2 \mid x^2 + y^2 = 1\}$. Thus we obtain the result. □

This proof consists of three parts. Part (1) is a construction in HoTT. Many libraries such as UniMath [38], Agda-UniMath [59], Cubical Agda [67], and the Rocq HoTT library [14] contain formalizations of this construction. Part (3) is a classical argument about topological realizations that could be stated using the Mathlib library [27] of Lean [55].

Our work focuses on mechanizing part (2), a problem that has received little attention so far. In this part, the HoTT construction from (1) is interpreted as an element of a model that is later related to classical objects in (3). The difficulty is that proof assistants catered towards expressing (1) and (3) are disconnected: we are not aware of any tools that allow users to interpret synthetic results in classical settings. This is unfortunate since alongside homotopy theory, synthetic methods have found use in geometry [18, 48], topology [26], and computer science [13, 43, 64]. One of their strengths is generality: part (1), for instance, implies analogous statements of part (2) in simplicial [47] and cubical [16] set models of HoTT. In this work, we present `SynthLean`, a proof assistant combining support for Martin-Löf type theories such as HoTT with support for their models in presheaf categories.

`SynthLean` is a component of `HoTTLean` [42], a formalization project aiming to develop models of homotopy type theory jointly with synthetic arguments internal to these models. The `HoTTLean` repository, which contains the `SynthLean` library, can be found at

github.com/sinhp/HoTTLean

Contributions. We formalize a concrete syntax for Martin-Löf type theories with Π, Σ, Id types and base constants (Section 2), as well as their *natural model semantics* in presheaf categories (Section 3 and Section 4). We show soundness of the syntax for this semantics in Theorem 3.1. We then use these components to implement a frontend for `SynthLean` (Section 5), including a certifying typechecker based on normalization-by-evaluation. We evaluate the system on simple theories.

We keep pen-and-paper definitions close to their formalizations, introducing both in parallel with discussions of design choices and encountered challenges. Some identifiers and all ✓ marks are hyperlinks to the associated code, also archived on Zenodo [56]. Section 3 and Section 4 assume familiarity with category theory. Section 5 can be read immediately after Section 2.

2 Martin-Löf type theories

`SynthLean` supports axiomatizations that can be phrased as *Martin-Löf type theories* (henceforth just "theories") with base constants, type universes, and Π, Σ, Id type formers. A thorough exposition is given by Angiuli and Gratzer [10]. In this section, we outline our definition of theories.

2.1 Syntax and typing rules

Reasoning about the syntax of type theories in foundations that, like Lean, do not support induction-recursion [33] or quotient-inductive-inductive types (QIITs) [9], is known to require low-level encodings. We choose a fairly standard presentation in terms of proof-irrelevant relations on top of a grammar of raw expressions. We begin by fixing a set χ of base constant names; these will allow the syntax to refer to elements of the model (see Section 3.3). We then define raw expressions over χ (`SynthLean.Expr` χ in the formalization)

$$\begin{aligned}
t, u, v, w, A, B \in \mathsf{Expr} ::= \\
& c_A \mid x \mid \Pi_{\ell,\ell'} A.\, B \mid \lambda_{\ell,\ell',A}.\, t \mid \mathsf{app}_{\ell,\ell',B}(t, u) \mid \\
& \Sigma_{\ell,\ell'} A.\, B \mid \mathsf{pair}_{\ell,\ell',B}(t, u) \mid \mathsf{fst}_{\ell,\ell',A,B}\, t \mid \mathsf{snd}_{\ell,\ell',A,B}\, t \mid \\
& \mathsf{Id}_{\ell,A}(t, u) \mid \mathsf{refl}_\ell\, t \mid \mathsf{idRec}_{\ell,\ell'}(t, A, u, v, w) \mid \\
& U_\ell \mid \mathsf{El}\, t \mid \mathsf{code}\, A
\end{aligned}$$

where $c \in \chi$ are constant names, $x \in \mathbb{N}$ are variables represented as De Bruijn indices, and $\ell, \ell' \in \mathbb{N}$ are universe levels. We consistently use uppercase A, B, C for types and lowercase t, u, v, w for terms, though the grammar does not formally differentiate between them.

Readers familiar with type theory may notice that our expressions are heavily annotated with types and universe levels. There are two reasons for this: firstly, this makes type synthesis trivial, which we exploit in our proof of uniqueness of typing (Theorem 2.4); secondly, it permits defining the interpretation function (in Section 3.3) by structural recursion on raw expressions.

The next step is to formalize *simultaneous substitution*, the operation that replaces all variables in an expression by other expressions. We follow the battle-tested approach of Autosubst [61]. Here, one defines raw substitutions as maps $\sigma : \mathbb{N} \to \mathsf{Expr}$, and their action as a total function given by recursion on expressions. This action computes through all expression formers, e.g. $(\mathsf{code}\, A)[\sigma] = \mathsf{code}\,(A[\sigma])$. One then proves a number of equations which, when viewed as rewrite rules, form a normalizing system that decides the equational theory of substitutions. We register this system with the auto-rewriting tactic `simp`, producing a macro `autosubst` standing for `simp only [autosubst]`.

Besides expressions, we have *raw contexts* and *raw theories*. Raw contexts (`SynthLean.Ctx` χ) are lists $(\mathsf{Expr}\times\mathbb{N})^*$ of binder types along with their universe levels. We use capital Greek letters Γ, Δ for contexts. A raw theory (`SynthLean.Axioms` χ) is a partial map $\mathbb{T} : \chi \rightharpoonup \mathsf{Expr} \times \mathbb{N}$ from base constant names to their types and universe levels. It is intended to be defined at exactly those names that correspond to axioms of the theory. This is sometimes called a *signature*, with the word "theory" reserved for a signature together with a set of equations over terms in the signature. However, we do not support such equations; they can make typechecking in the theory undecidable [24], prohibiting us from reusing Lean's elaborator. Thus, we identify signatures with theories. We write $\cdot$ for the empty context or theory, $\Gamma.A_\ell$ for a context Γ extended by a type A at level ℓ, and $\mathbb{T}, c :_\ell A$ for a theory $\mathbb{T}$ extended by a mapping from $c \in \chi$ to a type A at level ℓ.

With these ingredients, we may now specify the five *judgments* of a theory: relations that determine which raw expressions are well-formed, and which are *judgmentally equal*. Judgments are all of the form $\mathbb{T} \mid \mathcal{J}$, read as "theory $\mathbb{T}$ proves that $\mathcal{J}$". Where the theory can be inferred from context, we drop the prefix $\mathbb{T} \mid$ and just write $\mathcal{J}$. We have

- $\mathbb{T} \mid \Gamma\ \mathsf{cx}$ meaning "(theory $\mathbb{T}$ proves that) Γ is a well-formed context" (`WfCtx` in Lean)
- $\mathbb{T} \mid \Gamma \vdash_\ell A\ \mathsf{type}$ meaning "A is a well-formed type at level ℓ in context Γ" (`WfTp`)
- $\mathbb{T} \mid \Gamma \vdash_\ell t : A$ meaning "t is a well-formed term of type A at ℓ in Γ" (`WfTm`)
- $\mathbb{T} \mid \Gamma \vdash_\ell A \equiv B\ \mathsf{type}$ meaning "A and B are judgmentally equal types at ℓ in Γ" (`EqTp`)
- $\mathbb{T} \mid \Gamma \vdash_\ell t \equiv u : A$ meaning "t and u are judgmentally equal terms of type A at ℓ in Γ" (`EqTm`)

We say that a raw theory $\mathbb{T}$ is *well-formed*, written $\mathbb{T}$ thy (`Axioms.Wf`), when for every $c \in \chi$, $A \in \mathsf{Expr}$, $\ell \in \mathbb{N}$ such that $\mathbb{T}(c) = (A, \ell)$, one has $\mathbb{T} \mid \cdot \vdash_\ell A$. In words, the type of every base constant must be well-formed in the empty context with respect to the theory. Restricting to only closed constants is an important simplification: it makes the action of substitutions on constants trivial as there is only one substitution for the empty context.

The typing judgments are defined by mutual induction as the least relations closed under rules of inference. Most rules are standard; below, we illustrate our design choices and departures from common presentations.

Universe hierarchy. Since we ultimately wish to build models of arbitrary theories, we must observe some Gödelian restrictions. Lean's impredicative universe `Prop` of proof-irrelevant propositions has high proof-theoretic strength, and would allow us to establish consistency of a countable hierarchy of predicative universes [58]. However, in order to support constructions such as that in Theorem 1.1, we need our models to carry proof-relevant data. In other words, we must build them in `Type` rather than `Prop`. Since Lean does not support quantification over universe levels, we can only support a finite hierarchy of universes U_ℓ with $\ell < \ell_{\max}$. Model constructions will typically define objects in `Type u` polymorphically over `u`, and then instantiate them at a finite sequence `u = 0, u = 1, ...`.

As a technical device that simplifies the formalization, we also annotate typing judgments, binder types in a context, and base constants in a theory by their universe levels. Universe-related rules are shown below.

$$\frac{}{\cdot\ \mathsf{cx}} \qquad \frac{\Gamma\ \mathsf{cx} \qquad \Gamma \vdash_\ell A\ \mathsf{type}}{\Gamma.A_\ell\ \mathsf{cx}} \qquad \frac{\Gamma\ \mathsf{cx} \qquad \ell < \ell_{\max}}{\Gamma \vdash_{\ell+1} U_\ell\ \mathsf{type}}$$

$$\frac{\Gamma \vdash_{\ell+1} t : U_\ell}{\Gamma \vdash_\ell \mathsf{El}\, t\ \mathsf{type}} \qquad \frac{\ell < \ell_{\max} \qquad \Gamma \vdash_\ell A\ \mathsf{type}}{\Gamma \vdash_{\ell+1} \mathsf{code}\, A : U_\ell}$$

Similarity to Lean. In Section 5.2, we translate expressions in Lean's type theory [22] into expressions in our system. For the translation to be sound, typing derivations of those Lean expressions need to have corresponding derivations in `SynthLean`. To simplify the translation, we choose inference rules that closely mimick those of Lean, whenever possible. One such choice is in the formation of Π (and Σ) types, shown below. Their formation is *universe-polymorphic*: the universe level of a dependent product (or sum) type is the maximum of the universe levels of A and B. This forces a semantic construction described in Section 4.3.

$$\frac{\Gamma \vdash_\ell A\ \mathsf{type} \qquad \Gamma.A_\ell \vdash_{\ell'} B\ \mathsf{type}}{\Gamma \vdash_{\max(\ell,\ell')} \Pi A.\, B\ \mathsf{type}}$$

One departure from Lean is in our treatment of universes: above, we have presented a Coquand-style system [28] with El and code, whereas Lean implements the Russell style that makes no distinction between the syntax of types and of type codes. To ensure soundness of translation, we postulate that types and type codes are in bijection.

$$\frac{\Gamma \vdash_{\ell+1} t : U_\ell}{\Gamma \vdash_{\ell+1} \mathsf{code}\,(\mathsf{El}\, t) \equiv t : U_\ell} \qquad \frac{\ell < \ell_{\max} \qquad \Gamma \vdash_\ell A\ \mathsf{type}}{\Gamma \vdash_\ell \mathsf{El}\,(\mathsf{code}\, A) \equiv A\ \mathsf{type}}$$

Semantic considerations. Besides simplifying the translation, mimicking the typing rules of Lean can also aid model constructions (Section 3). This is because, in general, it can be easier to build models that interpret object-level features as the corresponding meta-level features as long as their behaviors are sufficiently similar. For example, to construct a groupoid model of type theory, `SynthLean` Σ-types could be interpreted using Lean's native Σ-types. This works because of the universe-polymorphic rule given above.

In Tarski-style systems, it is common to also include formers for type codes such as $\underline{\sigma}(t, u)$ where

$$\frac{\Gamma \vdash t : U_\ell \qquad \Gamma.(\mathsf{El}\, t) \vdash u : U_\ell}{\Gamma \vdash \mathsf{El}\, \underline{\sigma}(t, u) \equiv \Sigma(\mathsf{El}\, t).\, (\mathsf{El}\, u)\ \mathsf{type}}$$

We do not have code formers such as σ: the only way to form codes is by using code on a type. This is because, in models such as the one described above, the above equation amounts to saying that `Σ (a : ULift A), ULift (B a.down) = ULift (Σ (a : A), B a)` holds in Lean, where `ULift` is the universe lifting operation. Unfortunately, this is not provable (though a bijection can be constructed).

Presuppositions. The question expressed by a judgment is only well-posed if certain *presuppositions* are met: given a raw type A and raw context Γ, it doesn't make sense to ask whether $\Gamma \vdash A$ type if Γ is not itself known to be well-formed. Formally, this means the theory should enjoy *inversion* metatheorems (Theorem 2.2). Inversion is more or less difficult to prove depending on choices made in the presentation of syntax. In `logrel-mltt` [4], for example, it follows from a complex logical relations argument. Following ideas of Bauer et al. [15], we make inversion provable by a simple induction on typing derivations by building in presuppositions as additional assumptions in the inference rules. For example, the introduction rule for identity types below has an additional grayed out well-typedness assumption. We later prove that the same rule without the assumption is admissible.

$$\frac{\Gamma \vdash_\ell A \qquad \Gamma \vdash_\ell t : A \qquad \Gamma \vdash_\ell u : A}{\Gamma \vdash_\ell \mathsf{Id}_{\ell,A}(t,u)}$$

Proof irrelevance. We have formalized typing judgments $\Gamma \vdash_\ell t : A$ as proof-irrelevant relations in the `Prop` universe. This simplifies the formalization by quotienting away the details of particular typing derivations. However, Lean imposes limitations on how irrelevant proofs can be eliminated: it is not possible to construct data (i.e., terms whose types live in `Type`) by recursion on proofs. In particular, we may not define the interpretation function (see Section 3.3) by recursion on typing derivations; instead, we proceed by recursion on raw expressions. This forces us to annotate expressions with additional type information needed by the interpretation function.

Alternative designs.

- A common approach in semantics literature is to use *explicit substitutions* [1], meaning to turn the action of substitution from a function defined by recursion on syntax into additional type and term formers, quotienting by the equations they should obey. Although we believe this construction should be definable using Lean's quotient types, similar definitions are awkward to work with in practice [46].
- A *PER-style* presentation of typing rules, such as in Lean4Lean [22] presents only judgmental equalities. Then an object is well-formed when it is equal to itself as in $\Gamma \vdash t : A \triangleq \Gamma \vdash t \equiv t : A$. This gives an economical definition, but we have found it to complicate inversion.
- Instead of proving admissibility of substitution (Theorem 2.1), it is possible to postulate it as an inference rule. Unfortunately, this also would complicate our proof of inversion.

2.2 Syntactic metatheory

We now establish basic facts about the syntactic behavior of theories and discuss their formalization. A raw substitution $\sigma : \mathbb{N} \to \mathsf{Expr}$ is said to be *well-formed* from Δ to Γ, written $\Delta \vdash \sigma : \Gamma$ (`WfSb.mk`), when Δ cx, Γ cx, and for every variable $x \in \mathbb{N}$ such that $\Gamma \vdash_\ell x : A$, we have $\Delta \vdash_\ell \sigma(x) : A[\sigma]$. The action $t[\sigma]$ of substitutions σ on expressions t extends to an action $\mathcal{J}[\sigma]$ on judgments $\mathcal{J}$ in the obvious way.

Theorem 2.1 (Admissibility of substitution ✓)**.** *If* $\Gamma \vdash_\ell \mathcal{J}$ *and* $\Delta \vdash \sigma : \Gamma$*, then* $\Delta \vdash_\ell \mathcal{J}[\sigma]$*.*

Proof. We first prove the analogous fact for well-formed renamings, and then proceed by mutual induction on typing derivations. A number of auxiliary definitions building in presuppositions must be made for the induction to go through. ◻

Currently, support for mutual induction in Lean is poor: the system generates primitive recursors for mutually inductive types, but the `induction` tactic does not know about these. We have written a simple `mutual_induction` tactic that can build the necessary low-level scaffolding. Many of the inductive cases can be discharged automatically by the built-in `grind` tactic.

Theorem 2.2 (Inversion ✓)**.** *If* $\Gamma \vdash_\ell \mathcal{J}$ *then* Γ cx*, if* $\Gamma \vdash_\ell t : A$ *then* $\Gamma \vdash_\ell A$ type*, and similarly for the other judgments.*

Proof. By mutual induction on typing derivations, using Theorem 2.1. ◻

As corollaries of substitution and inversion, we establish inversion lemmas for all term and type formers. For example

Corollary 2.3 (Inversion of refl ✓)**.** *If* $\Gamma \vdash_{\ell_0} \mathsf{refl}_\ell\, t : C$*, then* $\ell_0 = \ell$ *and there exists* A *such that* $\Gamma \vdash_\ell t : A$ *and* $\Gamma \vdash_\ell C \equiv \mathsf{Id}_{\ell,A}(t,t)$ type*.*

We also show that a number of presupposition-free inference rules are admissible. For example, formation of identity types can now drop the extra typing assumption.

$$\frac{\Gamma \vdash_\ell t : A \qquad \Gamma \vdash_\ell u : A}{\Gamma \vdash_\ell \mathsf{Id}_{\ell,A}(t,u)}$$

Our presentation does not enjoy inversion for theories, i.e., $\mathbb{T} \mid \Gamma \vdash \mathcal{J}$ does not imply $\mathbb{T}$ thy. Instead, we carry the latter around as an additional assumption in the formalization. This is because building the property in using presuppositions forces $\mathbb{T}$ to be finitely supported, yet we wish to support infinite theories.

Theorem 2.4 (Unique levels and types ✓)**.** *If* $\Gamma \vdash_\ell t : A$ *and* $\Gamma \vdash_{\ell'} t : A'$*, then* $\ell = \ell'$ *and* $\Gamma \vdash_\ell A \equiv A'$ type*.*

Proof. Level and type inference procedures (`synthLvl`, `synthTp`) can be defined by structural recursion on raw expressions. We show by induction that $\ell =$ `synthLvl Γ t` and that $\Gamma \vdash_\ell A \equiv$ `synthTp Γ t` type. The same relations hold of ℓ' and A'.

□

Finally, we conjecture that type formers are injective and add this as a Lean `axiom`. Proofs of this property are known for very similar systems [50]. They are, however, laborious, and we consider formalizing one outside the scope of this work. The soundness argument in Section 3 does not depend on this axiom, but the certifying typechecker uses it to output certificates of evaluation and judgmental equality (Section 5.3).

Conjecture 2.5 (Injective type formers).
If $\Gamma \vdash_\ell \Pi_{\ell',\ell''} A.B \equiv \Pi_{\ell',\ell''} A'.B'$ type*, then* $\Gamma \vdash_{\ell'} A \equiv A'$ type *and* $\Gamma.A_{\ell'} \vdash_{\ell''} B \equiv B'$ type*. Analogously for* Σ *and* Id *types.*

3 Natural model semantics of ML theories

In this section, we introduce the categorical structures used to model Martin-Löf type theories, collectively referred to as *natural model semantics* [11]. The natural model approach is equivalent to using *categories with families* (CwFs) [32] while being more concise, and allowing us to exploit general category-theoretic results from Mathlib.

We prove that our semantics soundly models the syntax, providing a foundation for the model-theoretic mode of `SynthLean`. This means every well-formed context, term, type, and substitution, has a corresponding semantic entity. The semantics is furthermore *strict* [40], meaning that judgmentally equal types and terms correspond to equal (not just isomorphic) semantic structures.

All this data is packaged together in *interpretation functions* $[\![-]\!]$ for well-formed contexts, substitutions, types, and terms. Interpretation is fairly modular: each kind of syntactic entity corresponds to a distinct semantic construction. We introduce this in stages: first just enough semantics to specify the signatures of interpretation functions and state soundness, then the semantics of further syntactic features.

3.1 Contexts and universes

Fix a category Ctx with a terminal object, so named because well-formed contexts Γ are interpreted as objects $[\![\Gamma]\!] \in \mathsf{Ctx}$, and well-formed substitutions $\Delta \vdash \sigma : \Gamma$ as morphisms $[\![\sigma]\!] : [\![\Delta]\!] \to [\![\Gamma]\!]$ there. For this reason, we refer to arbitrary objects in Ctx as *semantic contexts*, and arbitrary maps there as *semantic substitutions*. We use boldface for semantic contexts, substitutions, terms, and types, e.g. bold Greek $\boldsymbol{\Delta}, \boldsymbol{\Gamma}$ for semantic contexts, and $\boldsymbol{\sigma}, \boldsymbol{\tau} : \boldsymbol{\Delta} \to \boldsymbol{\Gamma}$ for semantic substitutions. The terminal object $\mathbf{1}$ models the empty context.

Write $\mathbf{Psh}(\mathsf{Ctx})$ for the category of presheaves on Ctx, and $\mathbf{y} : \mathsf{Ctx} \to \mathbf{Psh}(\mathsf{Ctx})$ for the Yoneda embedding. Care must be taken here with regards to (Lean) universe levels: we set `Psh Ctx := Ctx`^{op} `⥤ Type v` where `v` is the level of the type of morphisms in Ctx. We will later need the fact that $\mathbf{Psh}(\mathsf{Ctx})$ is locally cartesian closed; since Mathlib can only prove this under the constraint that Ctx is a `SmallCategory` (namely one whose objects and morphisms live in the same universe), we impose this constraint.

Types and terms are interpreted as structures in $\mathbf{Psh}(\mathsf{Ctx})$. A *natural model universe* M (henceforth just "universe") consists of two presheaves Tm and Ty in $\mathbf{Psh}(\mathsf{Ctx})$ together with a natural transformation $\mathsf{tp} : \mathsf{Tm} \to \mathsf{Ty}$ which must be *representable* (defined below). To support a finite hierarchy of universes in the syntax, we have a semantic universe M_ℓ at each level $\ell \leq \ell_{\max}$. We write $M_\ell.\mathsf{Tm}$, $M_\ell.\mathsf{Ty}$, and $M_\ell.\mathsf{tp}$ for the three components, omitting M when it can be inferred from context.

The components of a universe provide a model for types and terms, with tp understood to project out the type of a term. For any level ℓ and semantic context $\boldsymbol{\Gamma}$, we define a *semantic type* $\boldsymbol{A}$ to be a map $\boldsymbol{A} : \mathbf{y}\boldsymbol{\Gamma} \to M_\ell.\mathsf{Ty}$, and a *semantic term* $\boldsymbol{t}$ *of type* $\boldsymbol{A}$ to be a map $\boldsymbol{t} : \mathbf{y}\boldsymbol{\Gamma} \to M_\ell.\mathsf{Tm}$ satisfying $M_\ell.\mathsf{tp} \circ \boldsymbol{t} = \boldsymbol{A}$. We will define interpretation functions such that if $\Gamma \vdash_\ell A$ then $[\![A]\!]$ is a semantic type in context $[\![\Gamma]\!]$, and if $\Gamma \vdash_\ell t : A$ then $[\![t]\!]$ is a semantic term of type $[\![A]\!]$.

$$\begin{tikzcd} & M_\ell.\mathsf{Tm} \arrow[d, "M_\ell.\mathsf{tp}"] \\ \mathbf{y}[\![\Gamma]\!] \arrow[ur, "{[\![t]\!]}"] \arrow[r, "{[\![A]\!]}"'] & M_\ell.\mathsf{Ty} \end{tikzcd}$$

The representability condition on tp is used to interpret context extension. It requires that for any object $\boldsymbol{\Gamma} \in \mathsf{Ctx}$ and map $\boldsymbol{A} : \mathbf{y}\boldsymbol{\Gamma} \to \mathsf{Ty}$, there is an object $\boldsymbol{\Gamma}.\boldsymbol{A} \in \mathsf{Ctx}$ such that $\mathbf{y}(\boldsymbol{\Gamma}.\boldsymbol{A})$ is the apex of a pullback cone on $\boldsymbol{A}$ and tp. In particular, for $\boldsymbol{\Gamma} = [\![\Gamma]\!]$ and $\boldsymbol{A} = [\![A]\!]$, $\boldsymbol{\Gamma}.\boldsymbol{A} = [\![\Gamma.A_\ell]\!]$ models the extended context:

$$\begin{tikzcd} \mathbf{y}[\![\Gamma.A_\ell]\!] \arrow[r, "\mathsf{var}_{[\![A]\!]}"] \arrow[d, "\mathbf{y}(\mathsf{disp}_{[\![A]\!]})"'] \arrow[dr, phantom, "\lrcorner", very near start] & M_\ell.\mathsf{Tm} \arrow[d, "M_\ell.\mathsf{tp}"] \\ \mathbf{y}[\![\Gamma]\!] \arrow[r, "{[\![A]\!]}"'] & M_\ell.\mathsf{Ty} \end{tikzcd}$$

These conditions are expressed in Lean as

```
structure Universe where
  Tm : Psh Ctx
  Ty : Psh Ctx
  tp : Tm ⟶ Ty
  ext {Γ : Ctx} (A : y(Γ) ⟶ Ty) : Ctx
  disp {Γ} (A : y(Γ) ⟶ Ty) : ext A ⟶ Γ
  var {Γ} (A : y(Γ) ⟶ Ty) : y(ext A) ⟶ Tm
  disp_pullback {Γ : Ctx} (A : y(Γ) ⟶ Ty) :
    IsPullback (var A) ym(disp A) tp A
```

Note that we do not merely state that a pullback exists and pick one with Mathlib's `pullback` (using the axiom of choice), but provide an explicit choice `ext`. Besides being necessary to express that the pullback is representable (i.e., in the image of $\mathbf{y}(-)$), this is also more convenient for the user when instantiating a particular model such as the groupoid model of HoTT0: the user may have access to a particular construction of the pullback associated with that model. Limiting distinct choices of the pullback allows users to convert between the abstract universe API and their particular instantiation of a universe.

3.2 Universe lifts

A single universe M_ℓ can model judgments such as $\Gamma \vdash_\ell A$ type, but does not necessarily contain a *syntactic universe*: a type of small types. We use *universe lifts* to model these. A universe lift from universe M to universe N consists of a pair of pullback squares of the following form

$$\begin{tikzcd} M.\mathsf{Tm} \arrow[r, "\mathsf{mapTm}"] \arrow[d, "M.\mathsf{tp}"'] \arrow[dr, phantom, "\lrcorner", very near start] & N.\mathsf{Tm} \arrow[d, "N.\mathsf{tp}"] \\ M.\mathsf{Ty} \arrow[r, "\mathsf{mapTy}"'] & N.\mathsf{Ty} \end{tikzcd} \qquad \begin{tikzcd} M.\mathsf{Ty} \arrow[r, "\mathsf{asTm}"] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & N.\mathsf{Tm} \arrow[d, "N.\mathsf{tp}"] \\ \mathbf{y1} \arrow[r, "U"'] & N.\mathsf{Ty} \end{tikzcd}$$

The first square says that terms and types in M can be lifted to terms and types in N. The second says that M.Ty can be viewed as a closed type U in N. This is called `UHom M N` in the formalization.

```
structure UHom (M N : Universe Ctx) where
  mapTm : M.Tm ⟶ N.Tm
  mapTy : M.Ty ⟶ N.Ty
  pb : IsPullback mapTm M.tp N.tp mapTy
  U : y(1_ Ctx) ⟶ N.Ty
  asTm : M.Ty ⟶ N.Tm
  U_pb : IsPullback asTm toTerminal N.tp U
```

A *universe sequence* containing $n + 1$ universes and n lifts between them is the following structure

```
structure UHomSeq where
  len : Nat
  objs (i : Nat) {h : i < len + 1} : Universe Ctx
  homSucc' (i : Nat) (h : i < len) :
    UHom (objs i) (objs (i + 1))
```

A model of any `SynthLean` theory must provide such a sequence of length at least $\ell_{\max}$. The field `homSucc'` stores lifts L_ℓ from M_ℓ to $M_{\ell+1}$. We set $[\![U_\ell]\!] = L_\ell.U$, $[\![\mathsf{code}\,A]\!] = L_\ell.\mathsf{asTm} \circ [\![A]\!]$, and define $[\![\mathsf{El}\,t]\!]$ via the universal property of the second pullback. The same property guarantees that El and code are mutually inverse.

Note also that universe lifts are not required to preserve any type or term formers. In the terminology of Kovács [49], this makes them *family morphisms* but not *family inclusions*. Our universe sequences are therefore not *family diagrams*.

3.3 Theories and soundness

We omitted a number of details in the above exposition. Firstly, defining interpretation functions in Lean is tricky due to their dependently-typed and recursive nature. Mathematically, the type of $[\![A]\!]$ as used above is $\hom(\mathbf{y}[\![\Gamma]\!], M_\ell.\mathsf{Ty})$. However, this would be inductive-recursive: the interpretation of contexts relies on context extension, which in turn relies on the interpretation of types. Following Streicher [65, ch. III], we untie the knot by turning the semantic context intending to model Γ into an input for two partial maps defined for $\mathbf{\Gamma} \in \mathsf{Ctx}$ and $\ell \in \mathbb{N}_{<\ell_{\max}}$ by structural recursion on raw expressions

- (`ofType`) $[\![-]\!]_{\mathbf{\Gamma},\ell} : \mathsf{Expr} \rightharpoonup \hom(\mathbf{y\Gamma}, M_\ell.\mathsf{Ty})$; and
- (`ofTerm`) $[\![-]\!]_{\mathbf{\Gamma},\ell} : \mathsf{Expr} \rightharpoonup \hom(\mathbf{y\Gamma}, M_\ell.\mathsf{Tm})$.

We then define a partial interpretation

$$[\![-]\!] : (\mathsf{Expr} \times \mathbb{N})^* \rightharpoonup \mathsf{Ctx}$$

of contexts (`ofCtx`) in terms of the above two. There are two sources of partiality: firstly, the given syntax of a type, term, or context need not be well-formed; secondly, the arbitrary semantic context $\mathbf{\Gamma}$ need not correspond to the correct syntactic one. We write $[\![x]\!]\downarrow$ to indicate that $[\![-]\!]$ is defined at x.

The second missing detail is the base case: a semantics for the base constants of a theory. A *raw base interpretation* (`Interpretation`) is a partial map

$$I : \chi \to (\ell : \mathbb{N}_{<\ell_{\max}}) \rightharpoonup \hom(\mathbf{y1}, M_\ell.\mathsf{Tm})$$

We write $[\![-]\!]_I$ for interpretation functions that use I on the base constants (i.e., $[\![c_A]\!]_{I,\mathbf{1},\ell} = I(c)$), though we will mostly leave this index implicit. I is *well-formed for* $\mathbb{T}$ (`Interpretation.Wf`) when for every $c \in \chi$, $A \in \mathsf{Expr}$, $\ell \in \mathbb{N}$ such that $\mathbb{T}(c) = (A, l)$, one has $I(c)\downarrow$ and $M_\ell.\mathsf{tp} \circ I(c) = [\![A]\!]_{I,\mathbf{1},\ell}$ (so in particular, $[\![A]\!]_{I,\mathbf{1},\ell}\downarrow$).

We may now state the main theorem:

Theorem 3.1 (Soundness ✓✓). *Given a base interpretation I well-formed for $\mathbb{T}$, we have*

- *If* $\mathbb{T} \mid \Gamma$ cx, *then* $[\![\Gamma]\!]_I \downarrow$.
- *If* $\mathbb{T} \mid \Gamma \vdash_\ell A$ type, *then* $[\![A]\!]_{I,[\![\Gamma]\!]_I,\ell}\downarrow$.
- *If* $\mathbb{T} \mid \Gamma \vdash_\ell t : A$, *then* $[\![t]\!]_{I,[\![\Gamma]\!]_I,\ell}\downarrow$ *and* $M_\ell.\mathsf{tp} \circ [\![t]\!]_{I,[\![\Gamma]\!]_I,\ell} = [\![A]\!]_{I,[\![\Gamma]\!]_I,\ell}$.
- *If* $\mathbb{T} \mid \Gamma \vdash_\ell A \equiv B$ type, *then* $[\![A]\!]_{I,[\![\Gamma]\!]_I,\ell} = [\![B]\!]_{I,[\![\Gamma]\!]_I,\ell}$.
- *If* $\mathbb{T} \mid \Gamma \vdash_\ell t \equiv u : A$, *then* $[\![t]\!]_{I,[\![\Gamma]\!]_I,\ell} = [\![u]\!]_{I,[\![\Gamma]\!]_I,\ell}$.

This result is established by, in essence, building a library of interpretations for every syntactic construct, and then putting this library together in a mutual induction on typing derivations. The formal proof is conceptually straightforward, though timeouts caused by its formidable size have forced us to split the proof into a number of small lemmas. The most important lemma is:

Lemma 3.2 (Semantic admissibility of substitution ✓✓). *For a well-formed substitution $\Delta \vdash \sigma : \Gamma$, we have:*

- *If* $[\![A]\!]_{[\![\Gamma]\!],\ell}\downarrow$, *then* $[\![A[\sigma]]\!]_{[\![\Delta]\!],\ell} = [\![A]\!]_{[\![\Gamma]\!],\ell} \circ \mathbf{y}[\![\sigma]\!]$.
- *If* $[\![t]\!]_{[\![\Gamma]\!],\ell}\downarrow$, *then* $[\![t[\sigma]]\!]_{[\![\Delta]\!],\ell} = [\![t]\!]_{[\![\Gamma]\!],\ell} \circ \mathbf{y}[\![\sigma]\!]$.

This result is at the heart of strict categorical semantics: since substitutions commute with term and type formers in the syntax, e.g. $(\Pi_{\ell,\ell'} A.\,B)[\sigma] = \Pi_{\ell,\ell'} A[\sigma].\,B[\sigma^\uparrow]$ (where $\sigma^\uparrow$ is σ weakened by one binder), they must also do so semantically. This is realized by *naturality* conditions, which constitute a significant portion of our formalization. Lemma

3.2 semantically validates Theorem 2.1 and allows one to interpret substitution as precomposition, for example

$$\frac{\Delta \vdash \sigma : \Gamma \qquad \Gamma \vdash_\ell t : A}{\Delta \vdash_\ell t[\sigma] : A[\sigma]} \qquad \begin{aligned} \mathsf{tp} \circ [\![t[\sigma]]\!] &= \mathsf{tp} \circ [\![t]\!] \circ \mathbf{y}[\![\sigma]\!] \\ &= A \circ \mathbf{y}[\![\sigma]\!] = [\![A[\sigma]]\!] \end{aligned}$$

4 Modeling type formers

An advantage of natural model semantics is their algebraic treatment of type constructors (Π, Σ, Id, etc.) in terms of *polynomial functors* [37, 54] (also called *containers* [2]). Our formalization is built on top of the Poly library [39] which describes the basic theory of these structures. We review what is needed for our purposes here.

4.1 Preliminaries on polynomial functors

Every map $f : E \longrightarrow B$ in a *locally cartesian closed* (LCC) category C induces a *polynomial endofunctor* $P_f : C \longrightarrow C$. We call f the *signature* of P_f. Polynomial functors are characterized by a *universal property* which states that the following two sets are in bijection, naturally in $\Gamma \in C^{\mathrm{op}}$ and $X \in C$:

- Maps of the form $\Gamma \longrightarrow P_f X$;
- Pairs $b : \Gamma \longrightarrow B$ and $g : E_b \longrightarrow X$, where E_b is a pullback of f along b.

$$\begin{tikzcd} X & E_b \arrow[l, "g"'] \arrow[r] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & E \arrow[d, "f"] \\ & \Gamma \arrow[r, "b"'] & B \end{tikzcd}$$

Hence, the value of a polynomial functor at X can be viewed as a sum $P_f X = \sum_{b:B} X^{E_b}$ of maps into X from the fibers of $f : E \longrightarrow B$.

We considered three ways of presenting this natural equivalence in Lean:

- Define the association $(\Gamma, X) \mapsto \hom(\Gamma, P_f X)$ as a profunctor, and the equivalence as a sequence of profunctor isomorphisms. This packages naturality properties as fields of an abstract isomorphism.
- Define a bijective function between the two sets of maps $\Gamma \longrightarrow P_f X$ and $\sum_{b:\Gamma \to B}(b^* f \longrightarrow X)$. Formalize naturalities in Γ and X as two or more additional theorems. This is the approach usually taken in Mathlib.
- Break up the second approach further: instead of using a bijection of hom sets, define functions that split a map $d : \Gamma \longrightarrow P_f X$ into a pair $\mathsf{fst}(d) : \Gamma \longrightarrow B$ and $\mathsf{snd}(d) : (\mathsf{fst}(d))^* f \longrightarrow X$, and in the other direction a function that packs such a pair into a single map. Formalize naturality laws as theorems about these maps.

The first approach is the neatest: formulating natural equivalences as isomorphisms allows us to compose equivalences, automatically obtaining naturality. This high-level argument is typical on paper, where elementary naturality proofs are usually omitted. This abstract definition often needs to be unfolded in order to obtain the underlying bijection of sets. Unfortunately, significant performance degradation caused by this unfolding made interactive proof development impossible and forced us to retreat to the more elementary second definition. With this approach, many naturality lemmas must be proven manually. Even this resulted in noticeably slow proofs, so that in the end we use the third approach, collecting definitions and lemmas that comprise the equivalence in a Lean namespace `UvPoly.Equiv`. For example, the function that pairs up two maps is shown below. Like `M.ext`, it allows providing a specific choice of pullback `R`.

```
def UvPoly.Equiv.mk' (b : Γ ⟶ B) (x : R ⟶ X)
  (H : IsPullback (P := R) f g b P.p) :
  Γ ⟶ P @ X := ...
```

Polynomial functors also enjoy a notion of composition. Given two maps $p : E \longrightarrow B$ and $q : D \longrightarrow A$ in an LCC category C, there is another polynomial functor denoted $q \triangleleft p$ which satisfies $P_{q \triangleleft p} \cong P_q \circ P_p$. Polynomial composition will play a role in the semantics of Σ-types (Section 4.2).

4.2 Universe-monomorphic Π and Σ types

Recall from Section 2.1 that, like Lean, our syntax puts dependent products and sums in the max of two universe levels. A priori, this requires model constructions to provide $O(\ell_{\max}^2)$ pieces of data, one for each pair (ℓ, ℓ'). Although this would work, it turns out to be possible to generically *lift* a sequence of $\ell_{\max}$ universe-monomorphic models of Π and Σ—that is ones which at every level ℓ validate the rule below—to form a polymorphic model. We implement this in order to simplify model constructions, proceeding in two steps: first axiomatize the monomorphic semantics, then lift it to two different levels in a model-independent manner.

$$\frac{\Gamma \vdash_\ell A \;\mathsf{type} \qquad \Gamma.A_\ell \vdash_\ell B \;\mathsf{type}}{\Gamma \vdash_\ell \Pi A.\, B \;\mathsf{type}}$$

The semantics of Π and Σ types can be concisely expressed using polynomial functors. For a given universe, the universal property of P_{tp} says that a map $(\boldsymbol{A}, \boldsymbol{B}) : \mathbf{y\Gamma} \longrightarrow P_{\mathsf{tp}}\mathsf{Ty}$ is the same as a semantic type $\boldsymbol{A} : \mathbf{y\Gamma} \longrightarrow \mathsf{Ty}$ and another $\boldsymbol{B} : \mathbf{y}(\mathbf{\Gamma}.\boldsymbol{A}) \longrightarrow \mathsf{Ty}$ depending on $\boldsymbol{A}$. This is exactly the input data for Π formation. Similarly, a map $(\boldsymbol{A}, \boldsymbol{t}) : \mathbf{y\Gamma} \longrightarrow P_{\mathsf{tp}}\mathsf{Tm}$ is the same a semantic term $\boldsymbol{t}$ that depends on $\boldsymbol{A}$, thus comprising the data of a function. A Π-*type structure* on a universe consists of two maps $\Pi : P_{\mathsf{tp}}\mathsf{Ty} \longrightarrow \mathsf{Ty}$ and $\lambda : P_{\mathsf{tp}}\mathsf{Tm} \longrightarrow \mathsf{Tm}$ in a pullback square

$$\begin{tikzcd} P_{\mathsf{tp}}\mathsf{Tm} \arrow[r, "\lambda"] \arrow[d, "P_{\mathsf{tp}}\mathsf{tp}"'] \arrow[dr, phantom, "\lrcorner", very near start] & \mathsf{Tm} \arrow[d, "\mathsf{tp}"] \\ P_{\mathsf{tp}}\mathsf{Ty} \arrow[r, "\Pi"'] & \mathsf{Ty} \end{tikzcd}$$

Composing $(\boldsymbol{A}, \boldsymbol{B})$ with Π produces a semantic type $\Pi \circ (\boldsymbol{A}, \boldsymbol{B}) : \mathbf{y}\boldsymbol{\Gamma} \to \mathsf{Ty}$. We set $[\![\Pi_{\ell,\ell} A.\, B]\!] = M_\ell.\Pi \circ ([\![A]\!], [\![B]\!])$ for A, B both at level ℓ. The map λ and the fact that the square commutes provide semantics for λ formation, whereas the universal property of the pullback square models function application, with β, η-reduction. Declaring Π-type structures in Lean is quite straightforward; we write `M.Ptp` for $P_{M.\mathsf{tp}}$.

```
structure Pi where
  Pi : M.Ptp.obj M.Ty ⟶ M.Ty
  lam : M.Ptp.obj M.Tm ⟶ M.Tm
  Pi_pullback :
    IsPullback lam (M.Ptp.map M.tp) M.tp Pi
```

To carry the proof of Theorem 3.2 through for Π, we must prove the appropriate naturality clauses. These clauses, in turn, rely on naturality of the universal property of $P_{M.\mathsf{tp}}$. We specialize the namespace `UvPoly.Equiv` to $P_{M.\mathsf{tp}}$ and explicit choices of pullback given by `M.ext` in another namespace, `PtpEquiv`. It contains the following:

```
def mk (A : y(Γ) ⟶ M.Ty)
  (B : y(M.ext A) ⟶ X) : y(Γ) ⟶ M.Ptp.obj X

lemma mk_comp_left (σ : Δ ⟶ Γ)
  (A : y(Γ) ⟶ M.Ty)  (σA) (eq : ym(σ) ≫ A = σA)
  (B : y(M.ext A) ⟶ X) :
  ym(σ) ≫ mk A B =
    mk σA (ym(M.substWk σ A σA eq) ≫ B)
```

The definition packs $\boldsymbol{A}$ and $\boldsymbol{B}$ into a map $(\boldsymbol{A}, \boldsymbol{B})$, whereas the lemma states that $(\boldsymbol{A}, \boldsymbol{B}) \circ \mathbf{y}\boldsymbol{\sigma} = (\boldsymbol{A} \circ \mathbf{y}\boldsymbol{\sigma}, \boldsymbol{B} \circ \mathbf{y}\boldsymbol{\sigma}^{\uparrow})$. Note that `mk_comp_left` takes an extra argument `σA` and a proof that `ym(σ) ≫ A = σA` instead of just using `ym(σ) ≫ A`. This is an instance of a general technique known as *fording* [25]. The problem is that the type of `B` in `mk` depends on `A`, and in general we may need to use a term whose type is provably, but not judgmentally, equal to `y(M.ext (ym(σ) ≫ A)) ⟶ X`. The fording transformation relaxes this requirement, allowing users of the theorem to provide a propositional equality. Since constructions in category theory are often dependently-typed, we are forced to use fording pervasively throughout the codebase. In other situations, we deal with type mismatches using the domain-specific `eqToHom` construct provided by Mathlib. This produces an arrow `Y ⟶ Y'` out of an equality `Y = Y'`, allowing one to compose two morphisms whose endpoints are provably, but again not judgmentally, equal.

To model Σ-types, we consider the polynomial composition $\mathsf{tp} \triangleleft \mathsf{tp}$ of tp with itself. The codomain of $\mathsf{tp} \triangleleft \mathsf{tp}$ is $P_{\mathsf{tp}}\mathsf{Ty}$, and we write $D_{\mathsf{tp},\mathsf{tp}}$ for its domain. $D_{\mathsf{tp},\mathsf{tp}}$ enjoys a universal property that exactly matches the inputs to the (again universe-monomorphic) pair formation rule:

$$\frac{\Gamma.A_\ell \vdash_\ell B \text{ type} \qquad \Gamma \vdash_\ell t : A \qquad \Gamma \vdash_\ell u : B[t]}{\Gamma \vdash_\ell \mathsf{pair}_{\ell,\ell,B}(t, u) : \Sigma_{\ell,\ell} A.\, B}$$

Namely, an arrow $(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{t}, \boldsymbol{u}) : \mathbf{y}\boldsymbol{\Gamma} \to D_{\mathsf{tp},\mathsf{tp}}$ is the same as $(\boldsymbol{A}, \boldsymbol{B}) : \mathbf{y}\boldsymbol{\Gamma} \to P_{\mathsf{tp}}\mathsf{Ty}$, a semantic term $\boldsymbol{t}$ of type $\boldsymbol{A}$, and $\boldsymbol{u}$ of type $\boldsymbol{B}[\boldsymbol{t}]$. A Σ-*type structure* on a universe then amounts to requiring two maps pair and Σ making the following a pullback square:

$$\begin{tikzcd} D_{\mathsf{tp},\mathsf{tp}} \arrow[r, "\mathsf{pair}"] \arrow[d, "\mathsf{tp} \triangleleft \mathsf{tp}"'] \arrow[dr, phantom, "\lrcorner", very near start] & \mathsf{Tm} \arrow[d, "\mathsf{tp}"] \\ P_{\mathsf{tp}}\mathsf{Ty} \arrow[r, "\Sigma"'] & \mathsf{Ty} \end{tikzcd}$$

As was the case for Π, the single pullback suffices to model introduction, elimination, and computation rules. In Lean, the required structure is

```
structure Sigma where
  Sig : M.Ptp.obj M.Ty → M.Ty
  pair : compDom (uvPolyTp M) (uvPolyTp M) ⟶ M.Tm
  Sig_pullback : IsPullback pair ((uvPolyTp
    M).compP (uvPolyTp M)) M.tp Sig
```

The data of a Π and Σ structure on each M_ℓ is attached to a universe sequence by means of two typeclasses

```
class PiSeq (s : UHomSeq Ctx) where
  nmPi (i : Nat) (ilen : i < s.len + 1) :
    Universe.Pi s[i]

class SigSeq (s : UHomSeq Ctx) where
  nmSig (i : Nat) (ilen : i < s.len + 1) :
    Universe.Sigma s[i]
```

4.3 Lifting Π and Σ types

We now come to the second step of our construction: lifting sums and products to operate across universe levels. For Π, we must construct $\Pi_{\ell,\ell'}$ and $\lambda_{\ell,\ell'}$ in pullback squares

$$\begin{tikzcd} P_{M_\ell.\mathsf{tp}} M_{\ell'}.\mathsf{Tm} \arrow[r, "\lambda_{\ell,\ell'}"] \arrow[d, "P_{M_\ell.\mathsf{tp}} M_{\ell'}.\mathsf{tp}"'] \arrow[dr, phantom, "\lrcorner", very near start] & M_m.\mathsf{Tm} \arrow[d, "M_m.\mathsf{tp}"] \\ P_{M_\ell.\mathsf{tp}} M_{\ell'}.\mathsf{Ty} \arrow[r, "\Pi_{\ell,\ell'}"'] & M_m.\mathsf{Ty} \end{tikzcd}$$

for every pair of universes $M_\ell, M_{\ell'}$ in a sequence, where $m \triangleq \max(\ell, \ell')$. This proof reaps the benefits of a high-level, abstract semantics. We simply paste the two pullback squares below onto the Π-type structure on M_m.

$$\begin{tikzcd} P_{M_\ell.\mathsf{tp}} M_{\ell'}.\mathsf{Tm} \arrow[r] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & P_{M_m.\mathsf{tp}} M_{\ell'}.\mathsf{Tm} \arrow[r] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & P_{M_m.\mathsf{tp}} M_m.\mathsf{Tm} \arrow[d] \\ P_{M_\ell.\mathsf{tp}} M_{\ell'}.\mathsf{Ty} \arrow[r] & P_{M_m.\mathsf{tp}} M_{\ell'}.\mathsf{Ty} \arrow[r] & P_{M_m.\mathsf{tp}} M_m.\mathsf{Ty} \end{tikzcd}$$

The pullbacks are constructed using general polynomial functor theory. The right square is the image of the composite universe lift from $M_{\ell'}$ to M_m under $P_{M_m.\mathsf{tp}}$, and it must be a pullback because polynomial functors preserve pullbacks. The left square is the naturality square at $M'_\ell.\mathsf{tp}$ of a *cartesian* natural transformation $P_{M_\ell.\mathsf{tp}} \to P_{M_m.\mathsf{tp}}$ corresponding to the composite lift from M_ℓ to M_m. We formalize the pasting as two definitions and a theorem.

```
def Pi : s[i].Ptp.obj s[j].Ty ⟶ s[max i j].Ty

def lam : s[i].Ptp.obj s[j].Tm ⟶ s[max i j].Tm

theorem Pi_pb : IsPullback (s.lam i j)
  (s[i].Ptp.map s[j].tp) s[max i j].tp(s.Pi i j)
```

Lifting Σ types proceeds similarly, but requires a more complex construction on $M_\ell.\mathsf{tp} \triangleleft M_{\ell'}.\mathsf{tp}$.

4.4 Identity types

Models of identity types are more involved than Π and Σ for two reasons. Firstly, they are formulated using several (weak) (pullback) squares rather than one. Secondly, although Id formation and refl only involve one universe level, the elimination rule shown below can target any other level ℓ'.

$$\frac{\begin{array}{c}\Gamma.(x : A_\ell).Id_{\ell,A}(t,x) \vdash_{\ell'} C \text{ type} \\ \Gamma \vdash_{\ell'} r : C[t, \mathsf{refl}_\ell\, t] \qquad \Gamma \vdash_\ell h : \mathsf{Id}_{\ell,A}(t,u)\end{array}}{\Gamma \vdash_{\ell'} \mathsf{idRec}_{\ell,\ell'}(t, A, u, r, h) : C[u, h]}$$

The structure for Id formation and refl introduction on a universe M consists of a kernel pair $k_1, k_2 : K \rightrightarrows \mathsf{Tm}$ as on the left below, and a commutative square as on the right.

$$\begin{tikzcd} K \arrow[r, "k_1"] \arrow[d, "k_2"'] \arrow[dr, phantom, "\lrcorner", very near start] & \mathsf{Tm} \arrow[d, "\mathsf{tp}"] \\ \mathsf{Tm} \arrow[r, "\mathsf{tp}"'] & \mathsf{Ty} \end{tikzcd} \qquad \begin{tikzcd} \mathsf{Tm} \arrow[r, "\mathsf{refl}"] \arrow[d, "\delta"'] & \mathsf{Tm} \arrow[d, "\mathsf{tp}"] \\ K \arrow[r, "\mathsf{Id}"'] & \mathsf{Ty} \end{tikzcd}$$

Here δ is the diagonal map defined using the universal property of K. A map $\mathbf{y\Gamma} \to K$ is therefore a pair of semantic terms of the same type. Composing with the map $\mathsf{Id} : K \to \mathsf{Ty}$ produces a map $\mathbf{y\Gamma} \to \mathsf{Ty}$ modeling the identity type between those two terms. Reflexivity proofs are modeled using the top map refl: for a semantic term $\boldsymbol{t} : \mathbf{y\Gamma} \to \mathsf{Tm}$, $\mathsf{refl} \circ \boldsymbol{t}$ is a term of type $\mathsf{Id}(\boldsymbol{t}, \boldsymbol{t})$. The typing judgment $\Gamma \vdash_\ell \mathsf{refl}_\ell\, t : \mathsf{Id}_{\ell,A}(t,t)$ is validated by the right square commuting. In Lean we have

```
structure IdIntro where
  K : Psh Ctx
  k1 : K ⟶ M.Tm
  k2 : K ⟶ M.Tm
  kp : IsKernelPair M.tp k1 k2
  Id : K ⟶ M.Ty
  refl : M.Tm ⟶ M.Tm
  refl_tp : refl ≫ M.tp =
    (kp.lift (1 M.Tm) (1 M.Tm) _) ≫ Id
```

Like in the ext field for universes, the user can choose a specific pullback K. Note also that we do not require the right-hand square above to be a pullback: doing so would model *extensional* type theory (uniqueness of identity proofs and equality reflection). As we are modeling *intensional* equality, we merely have a comparison map ρ between Tm and a chosen pullback I

$$\begin{tikzcd} \mathsf{Tm} \arrow[drr, bend left, "\mathsf{refl}"] \arrow[dr, dotted, "\rho"] \arrow[ddr, bend right, "\delta"'] & & \\ & I \arrow[r, "i_1"] \arrow[d, "i_2"] \arrow[dr, phantom, "\lrcorner", very near start] & \mathsf{Tm} \arrow[d, "\mathsf{tp}"] \\ & K \arrow[r, "\mathsf{Id}"'] & \mathsf{Ty} \end{tikzcd}$$

We record this second user-chosen pullback I in Lean as:

```
structure IdElimBase (ii : IdIntro M) where
  I : Psh Ctx
  i1 : I ⟶ M.Tm
  i2 : I ⟶ ii.k
  I_isPullback : IsPullback i1 i2 M.tp ii.Id
```

To model identity elimination, we use another construction involving polynomial functors, a variation of which can be found in the work of Awodey [12]. Consider $k_2 \circ i_2$ and $k_2 \circ \delta = \mathbf{1}_{\mathsf{Tm}}$ as signatures for polynomial functors. By the general theory of polynomial functors, the comparison map $\rho : \mathsf{Tm} \to I$ results in a natural transformation $\rho^* : P_{k_2 \circ i_2} \to P_{\mathbf{1}}$, which for another universe $N.\mathsf{tp}$ produces a naturality square:

$$\begin{tikzcd} M.\mathsf{Tm} \arrow[r, dotted, "\rho"] \arrow[d, "\mathbf{1}_{M.\mathsf{Tm}}"'] & i \arrow[dl, "k_2 \circ i_2"] \\ M.\mathsf{Tm} & \end{tikzcd} \qquad \begin{tikzcd} P_{k_2 \circ i_2} N.\mathsf{Tm} \arrow[r, "\rho^*_{N.\mathsf{Tm}}"] \arrow[d, "P_{k_2 \circ i_2} N.\mathsf{tp}"'] & P_{\mathbf{1}_{M.\mathsf{Tm}}} N.\mathsf{Tm} \arrow[d, "P_{\mathbf{1}_{M.\mathsf{Tm}}} N.\mathsf{tp}"] \\ P_{k_2 \circ i_2} N.\mathsf{Ty} \arrow[r, "\rho^*_{N.\mathsf{Ty}}"'] & P_{\mathbf{1}_{M.\mathsf{Tm}}} N.\mathsf{Ty} \end{tikzcd}$$

We say that *universe M has identity types that eliminate into universe N* when this naturality square is a weak pullback (that is, a pullback where comparison maps need not be unique). We formalize this as

```
structure Id' (N : Universe Ctx)
    extends IdElimBase M where
  weakPullback : WeakPullback
    (toIdElimBase.verticalNatTrans.app N.Tm)
    (toIdElimBase.iFunctor.map N.tp)
    ((UvPoly.id M.Tm).functor.map N.tp)
    (toIdElimBase.verticalNatTrans.app N.Ty)
```

Identity elimination is modeled by the weak pullback condition as follows. Assume a cone of the form:

$$\begin{tikzcd} \mathbf{y\Gamma} \arrow[r, "(\boldsymbol{t}, \boldsymbol{r})"] \arrow[d, "(\boldsymbol{t}, \boldsymbol{C})"'] & P_{\mathbf{1}_{M.\mathsf{Tm}}} N.\mathsf{Tm} \arrow[d, "P_{\mathbf{1}_{M.\mathsf{Tm}}} N.\mathsf{tp}"] \\ P_{k_2 \circ i_2} N.\mathsf{Ty} \arrow[r, "\rho^*_{N.\mathsf{Ty}}"'] & P_{\mathbf{1}_{M.\mathsf{Tm}}} N.\mathsf{Ty} \end{tikzcd}$$

The morphism $\mathbf{y\Gamma} \to P_{k_2 \circ i_2} N.\mathsf{Ty}$ can be identified with a semantic term $\boldsymbol{t} : \mathbf{y\Gamma} \to \mathsf{Tm}$ (of some type $\boldsymbol{A}$) and a motive $\boldsymbol{C} : \mathbf{y}(\boldsymbol{\Gamma}.(x : \boldsymbol{A}).(h : \mathsf{Id}_{\boldsymbol{A}}(\boldsymbol{t}, x))) \to N.\mathsf{Ty}$ for the eliminator (here we are using informal notation for clarity; the semantic context is really a series of ext applications). The morphism $\mathbf{y\Gamma} \to P_{\mathbf{1}_{M.\mathsf{Tm}}} N.\mathsf{Tm}$ can be identified with $\boldsymbol{t}$ (again) and a semantic term $\boldsymbol{r} : \mathbf{y\Gamma} \to N.\mathsf{Tm}$ whose type is the instantiated motive $\boldsymbol{C}[\boldsymbol{t}, \mathsf{refl}_{\boldsymbol{t}}]$. Then the weak pullback condition produces a map $(\boldsymbol{t}, \boldsymbol{c}) : \mathbf{y}\Gamma \longrightarrow P_{k_2 \circ i_2} N.\mathsf{Tm}$ where $\boldsymbol{c}$ is the map we want. This models idRec.

Identity data is attached to a universe sequence using the following typeclass

```
class IdSeq (s : UHomSeq Ctx) where
  nmII (i : Nat) {ilen : i < s.len + 1} :
    IdIntro s[i]
  nmIEB (i : Nat) {ilen : i < s.len + 1} :
    IdElimBase (nmII i)
  nmId (i j : Nat) {ilen : i < s.len + 1}
    {jlen : j < s.len + 1} : Id (nmIEB i) s[j]
```

5 The SynthLean proof assistant

The design and implementation of `SynthLean` is motivated by a need for the conveniences afforded by modern proof assistants. Although our formalization of the syntax and typing rules of theories already allows defining in Lean a theory $\mathbb{T}$, writing down a context Γ, term t, and type A, and stating that $\mathbb{T} \mid \Gamma \vdash t : A$, it is impractical to develop mathematics in $\mathbb{T}$ from this external perspective: writing down expressions in our type-annotated grammar is tedious, and proving that the judgment holds amounts to constructing the typing derivation by hand, a formidable task. Instead, we want to work internally in $\mathbb{T}$ by writing down t and A using high-level vernacular syntax, and having the typechecker automatically certify that $\mathbb{T} \mid \Gamma \vdash t : A$. To work efficiently, we also need tactics, type inference, storage for a library of definitions, and a responsive user interface. We now discuss how `SynthLean` provides these features, first by outlining an example usage of the prover, and then by covering how the implementation supports such usage.

5.1 Usage and interface

Users interact with the `SynthLean` frontend through *theory commands* and the *interpretation API*. Theory commands are used to state the base constants of a theory, and to make constructions in the theory. For example, let us declare a theory with four base constants: function extensionality, and three axioms describing a unit type.

```
declare_theory unitt
```

This command instructs the system to initialize a *theory environment*. Lean ordinarily keeps track of a single environment: an ordered list of axioms, definitions, and theorems. `SynthLean` extends this storage by a separate environment for each declared theory, to which we now append base constants.

```
unitt axiom funext {α : Type} {β : α → Type}
    (f g : (a : α) → β a) :
  (∀ a, Identity (f a) (g a)) → Identity f g

unitt axiom Unit : Type
unitt axiom u : Unit
unitt axiom uniq_u (u' : Unit) : Identity u' u
```

Each `unitt axiom` command has a dual effect: first, it extends the theory environment by the constant in question; second, it appends a *deeply embedded* representation of the axiom to the external Lean environment. We may observe both effects

```
unitt #print u -- Prints: axiom u : Unit
#print u
-- Prints: def u : CheckedAx [Unit] :=
--         { tp := .el Unit.val, ... }
```

The command `unitt #print u` prints `u` from the internal perspective: it appears as an ordinary axiom u : Unit. The command `#print u`, on the other hand, shows the external interpretation of `u` as a proof of $\mathsf{Unit} : U_0 \mid \cdot \vdash \boldsymbol{u} : \mathsf{El}\,\mathsf{Unit}$. The latter is stored as a structure in the type family `CheckedAx` indexed by a representation of the axioms that the type of the given axiom is well-formed with respect to; in this case, the empty environment extended by `Unit`.

```
structure CheckedAx (𝕋 : Axioms χ) where
  l : Nat -- Universe level of the type.
  tp : Expr χ -- Deeply embedded type.
  wf_tp : 𝕋 | [] ⊢[l] tp -- Typing derivation.
  ...
```

We may now use standard Lean tactics to prove that functions into the unit type are unique. The theory command below records a proof of

$$\mathsf{Unit} : U_0, \ldots \mid \cdot \vdash \lambda.(\cdots) : \\ \Pi(A : U_0)\,(f\,g : \mathsf{El}\,A \longrightarrow \mathsf{El}\,\mathsf{Unit}).\,\mathsf{Id}(f, g)$$

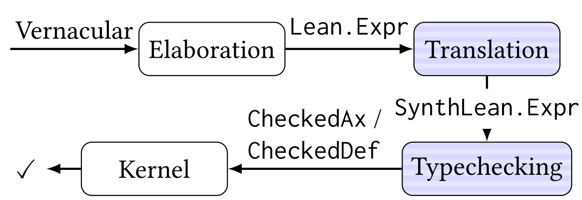
as a `CheckedDef`, a structure like `CheckedAx` that stores definitions rather than axioms.

```
unitt def uniq_fn {A : Type} (f g : A → Unit) :
    Identity f g := by
  apply funext; intro
  exact (uniq_u _).trans₀ (uniq_u _).symm₀
```

Having made definitions internally in the theory, we can use Theorem 3.1 to realize them in the natural model semantics.

5.2 Implementation: translation and typechecking

A theory definition command consumes high-level syntax (so-called *vernacular*) typed in by the user (possibly including tactic scripts) and proves the theory judgment $\mathbb{T} \mid \cdot \vdash t : A$, where t and A should reflect the vernacular term and type, respectively, and $\mathbb{T}$ is determined by axioms stored in the theory environment. The proof is recorded in Lean as a `CheckedDef`. This is achieved in four stages, illustrated in the diagram below.



In the first stage, vernacular syntax is *elaborated* into expressions `Lean.Expr` in the Lean type theory. For a definition, this produces a type and a term. Elaboration in Lean is a sophisticated process that includes type inference, running tactic scripts, and communicating interactive feedback to the user. Crucially, because our syntax mirrors that of Lean (Section 2.1), we may reuse built-in elaborators via the metaprogramming API [69]. Theory commands consequently support ordinary Lean syntax, and provide the same user experience as ordinary commands. This is an efficient way of obtaining the bulk of a proof assistant without implementing one ourselves. In the diagram, only components implemented by us are colored in.

Following elaboration, Lean expressions are translated into `SynthLean.Expr`s. This process is straightforward due to similarities between the two. Translation succeeds only on expressions that avoid unsupported features such as Lean's impredicative universe of propositions. This stage does not produce any proof certificates.

Next, `SynthLean.Expr`s are typechecked in order to produce the reflected typing derivation. The typechecker is *certifying* [57] rather than *certified*: we do not establish decidability of the typing relation, and do not define the checker as a total function in the Lean type theory. There is no completeness result, i.e., the checker is not guaranteed to succeed on every well-typed expression. Instead, we know only that every execution, should it succeed, outputs a proof of the judgment in question. This allows us to define the checker as a partial function, avoiding complex encodings of its call graph [20] that are usually needed to prove termination [70].

Finally, the typing derivation certificate is passed back to the Lean kernel to be checked for correctness. We now outline how certificates are produced.

5.3 Certification strategy

The typechecker is based on a normalization by evaluation (NbE) algorithm broadly similar to one proposed by Abel [3]. It consists of three components: the NbE evaluator, a judgmental equality (typed conversion) checker operating on NbE values, and type checking/synthesis procedures. We have inductively defined *values* (`Val` χ) to represent outputs of evaluation, *neutral forms* to represent expressions on which evaluation is stuck, and *defunctionalized closures* to represent binders. We can weaken values without traversing them by using De Bruijn levels instead of indices.

The soundness argument for evaluation and typechecking uses an inductive relation between values and terms or types. For a value vA and Γ, ℓ, A, we say that vA is *related* to A at ℓ in Γ, written $\Gamma \Vdash_\ell A \rightsquigarrow vA$ (`ValEqTp`), when subvalues of vA are evaluations of subterms of A. For example, $\Gamma \Vdash_{\max(\ell,\ell')} \Pi_{\ell,\ell'}A.\,B \rightsquigarrow \Pi_{\ell,\ell'}vA.\,cB$ just in case $\Gamma \Vdash_\ell A \rightsquigarrow vA$ and $\Gamma.A_\ell \Vdash_{\ell'} B \rightsquigarrow cB$, where cB is a closure. This simple relation turns out to be sufficient to certify results of evaluation and typechecking.

Each subroutine produces a proof that for the given input, if the preconditions are satisfied, so are the postconditions. Top-level type checking, for example, has signature

```
partial def checkTp (vΓ : Q(ValCtx String))
    (l : Q(Nat)) (T : Q(SynthLean.Expr String)) :
  TypecheckerM Q(∀ {Γ}, ValCtxEqCtx $𝕋 $vΓ Γ →
    $𝕋 | Γ ⊢[$l] $T)
```

The library `quote4` [34] is used here for type-safe quotation. A value of type `Q(T)` is a `Lean.Expr` whose type in the Lean type theory is `T`. We have set χ = `String`, representing base constant names as strings. The function takes (an expression representing) a `ValCtx` $v\Gamma$ (a context with binder types represented as NbE values), a universe level ℓ, and an expression T. It proves that if $\Gamma \rightsquigarrow v\Gamma$ (`ValCtxEqCtx`, defined from $- \Vdash - \rightsquigarrow -$ in the obvious way), then $\mathbb{T} \mid \Gamma \vdash_\ell T$ type.

This strategy produces certificates of linear size in the number of checker subroutine calls. One alternative would be to take proofs of preconditions as arguments and produce an unconditional proof of the postcondition, as in `checkTpAlt` below. However, if deployed without careful `let`-binding of subexpressions, this approach duplicates them in the output, resulting in certificates whose size grows superlinearly in the number of subroutine calls. Our approach does not require such management.

```
partial def checkTpAlt (vΓ : Q(TpEnv Lean.Name))
    (Γ : Q(Ctx String))
    (pfΓ : Q(ValCtxEqCtx $𝕋 $vΓ Γ))
    (l : Q(Nat)) (T : Q(Expr Lean.Name)) :
  TypecheckerM Q($𝕋 | Γ ⊢[$l] $T)
```

To satisfy their contracts, procedures in the typechecker output Lean proof terms by stitching together proofs output by subroutines. We illustrate one simple case: checking Π-types in `checkTp`.

```
match T with
| ~q(.pi $k $k' $A $B) => do
  let leq ← equateNat q($l) q(max $k $k')
  let Awf ← checkTp q($vΓ) q($k) q($A)
  let ⟨vA, vAeq⟩ ← evalTpId q($vΓ) q($A)
  let Bwf ←
    checkTp q(($vA, $k) :: $vΓ) q($k') q($B)
  return q(by as_aux_lemma =>
    introv vΓ; subst_vars
    apply WfTp.pi ($Bwf (vΓ.snoc ($vAeq vΓ ($Awf
   vΓ)))))
```

First, we ensure that $\ell = \max(k, k')$. We then check A, and evaluate it to vA (so that $\Gamma \Vdash_k A \rightsquigarrow vA$). Correctness of that evaluation is certified by `vAeq`. Finally, we check B in a context extended by vA and put the proofs together using the typing rule for Π. The `as_aux_lemma` macro ensures that the output of the tactic script is stored as a top-level lemma and checked by Lean only once rather than in every certificate.

Proofs produced by the checker rely on results of Section 2.2, including the injectivity axiom. One challenge here is reasoning about expressions modulo judgmental equality (recall, we do not use a QIIT presentation which would render them literally equal). Support for generalized rewriting (i.e., rewriting by non-equality relations) is limited, which in practice means that applying the conversion rule must often

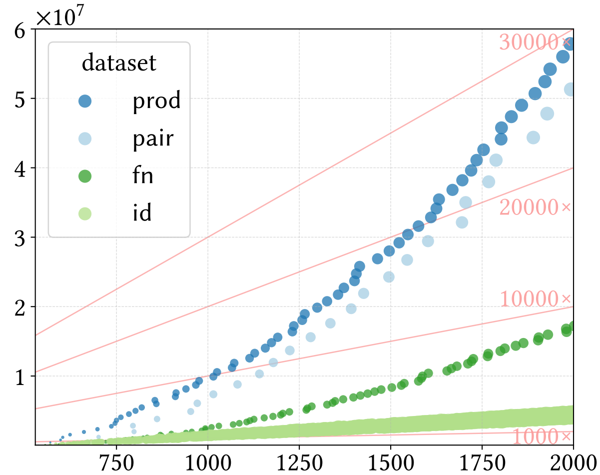


Figure 1. Lean kernel heartbeats (x-axis) vs `SynthLean` typechecker heartbeats (y-axis).

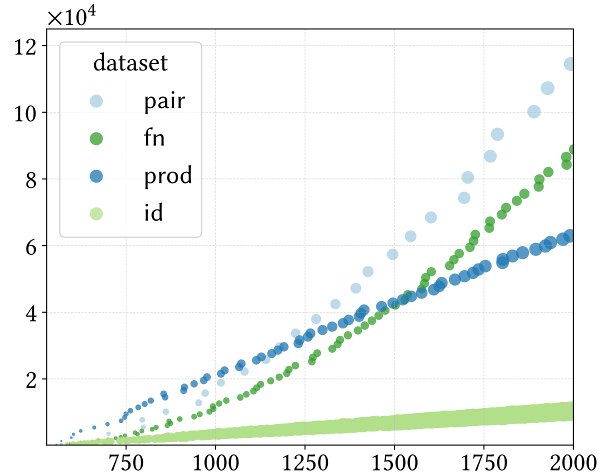


Figure 2. Lean kernel heartbeats (x-axis) vs `SynthLean` typing certificate graph size (y-axis).

be done manually. The `gcongr` tactic from Mathlib that automatically applies congruence lemmas is helpful, but much room for automation remains.

5.4 Experimental results

We conclude this section by empirically investigating the overhead incurred by our typechecker. We generate four synthetic datasets, each containing a series of theory definitions of increasing complexity: id, fn, prod, and pair. The id dataset contains N nested applications of the identity function to Type, with N increasing: Type, $(\lambda x.\, x)$ Type, $(\lambda x.\, x)$ $((\lambda x.\, x)$ Type), etc. Elements in fn are constant functions with N binders. The prod dataset contains N-ary dependent sums. Finally, pair consists of a sequence of functions, each taking an element x and returning an N-ary pair of x with itself.

We measured the performance of `SynthLean`'s four stages in terms of time taken, as well as the size of typing certificates produced by the typechecker. Time was measured in *heartbeats*, a deterministic, platform-independent counter provided by Lean that approximates the number of allocations made by a computation. Heartbeats correlate linearly with wall clock time, but are more reproducible and allow ignoring hardware details. We used Lean 4, version `v4.22.0-rc3`.

Typechecking dominates the runtime, so we focus on that. For every Lean expression e in a given dataset, Figure 1 plots (y) the time (in heartbeats) taken by our typechecker on e translated to `SynthLean`'s expression grammar against (x) the time taken by the Lean kernel to check e. Salmon lines show the overhead ratio $\frac{y}{x}$, and bubble size reflects the size of e. Our checker matches Lean kernel performance up to a constant on id but grows quadratically worse on other benchmarks. The linear behavior on id stems from memoization: without caching, it too had quadratic overhead. Despite the trend, the checker remains responsive during simple interactive development. We have not invested major effort into optimization, so there is likely room for asymptotic improvements.

Interestingly, although producing certificates takes time, in some cases they are highly compressible because they contain many identical subterms. We hash-cons [36] well-typedness certificates, i.e., maximally share their common subexpressions in memory, before handing them to the Lean kernel after the typechecking stage. This provides marginal speedups on some problems. Figure 2 plots (y) the number of memory cells occupied by typing certificates compressed via hash-consing against the same x-axis as in Figure 1. Certificates for prod take quadratically long to produce, but their compressed sizes grow linearly.

6 Related work

Mechanized metatheory of dependently-typed syntax. Much progress in this area has been driven by the desire to ensure correctness of proof assistant kernels. The MetaRocq project [62] contains a substantial formalization of the properties of PCUIC, an idealized version of the type theory of Rocq. Including an impredicative sort `Prop`, PCUIC has higher proof-theoretic strength than the theories we consider. The adjacent work of Adjedj et al. [6] considers a system similar to ours: MLTT with extensional Π and Σ, $\mathbb{N}$, intensional Id, and one universe. Both projects show that typechecking is decidable, producing typecheckers of the *certified* variety, in the sense of giving a once-and-for-all correctness proof (see Section 5.3). Unlike our approach, this generally requires complex termination arguments. Our light metatheory most closely resembles that developed in Rocq by Bauer et al. [15] for syntax in *paranoid* mode.

Analogues of our Conjecture 2.5 have been formally proven in Agda for MLTT with extensional Π-types, ℕ, and one universe by Abel et al. [4] via a logical relations argument (later adapted to Rocq by Adjedj and coauthors [op. cit.]), and for PCUIC in MetaRocq. The latter proof, relying on syntactic properties of untyped conversion, is not known to apply directly to presentations that, like ours, use a typed variant.

A major component of our machinery is certifying NbE. NbE for dependent type theory with one universe, extensional Π-types, and a unit type has been certified sound and complete by Wieczorek and Biernacki [70]. Here also, a termination argument is required.

The program transformation plugin of Boulier et al. [19] mechanically evaluates judgmental-equality-preserving maps from Rocq expressions to other expressions in the same system. Since it does not target a deep embedding of syntax, the plugin does not need to produce reflected certificates of typing derivations.

Certifying typecheckers. This approach was pioneered by Strub et al. [66], whose system produced Rocq certificates of the type-correctness of F* programs. Their type theory, being stratified into terms, types, and kinds with no type-level reduction, and relying on the soundness of an SMT solver for refinement typing, is not easily compared to ours. In opposition to our experiments, *checking* (in Rocq) rather than producing (in F*) certificates turned out to be a performance bottleneck.

More recently, Liesnikov and Cockx [53] presented a certifying typechecker for MLTT with extensional Π-types and parameterized (but not indexed) inductive types as part of the Agda Core effort. They use a Krivine machine [29] instead of NbE for reduction, and conversion is defined by what the machine does rather than as a standalone, undirected equational theory like in our work.

Initiality. Showing that an interpretation map as in Section 3.3 exists, if accompanied by a term model construction and a proof of the map's uniqueness, would establish that a particular choice of concrete syntax presents an *initial model* of the theory at hand. This was done in Agda by De Boer and Brunerie [30] following a strategy essentially identical to ours in the existence part, except for minor syntactic differences and a different choice of class of models: they target Cartmell's contextual categories [23] rather than natural models. Note that a term model or uniqueness of the interpretation map are not needed to transfer synthetic results to general models.

An initial model can be presented more succinctly as a QIIT [9], but this feature is unfortunately not available in Lean.

Categorical models of MLTT. The formalization of categorical models of type theory is an active area of research. Work here could be categorized according to whether it defines an abstract class of such models, or constructs a specific model (or both).

Our work on natural model semantics is an instance of the former. We are not aware of other mechanizations of this class of models; a synthetic treatment of type formers via polynomial functors on the category of types in HoTT was given in Agda by Aberlé and Spivak [5]. Equivalences between contextual categories, categories with families, natural models, and other kinds of models [7], though without semantic type formers, have been formalized in the univalent setting of UniMath [38].

As for *particular models*, the ones most relevant to our work are those that could be integrated with our system for synthetic reasoning. Du [31] formalized in Lean a simplicial model of type theory with Π-types, a single universe, and no univalence axiom, modulo a Kan-Quillen model structure on simplicial sets. Du phrased it as a contextual category. Building on this model in `SynthLean` may allow reasoning synthetically about Kan simplicial sets.

We have also discussed the groupoid model of type theory. Sozeau and Tabareau [63] worked towards a Rocq formalization, but this has not been completed. In univalent foundations, comprehension categories [44] and bicategories [8] behave better than categories with families. The groupoid model is formalized as an example comprehension category in the Unimath project [op. cit.], also without type formers.

Other related work. Some synthetic results in HoTT have been developed using Lean, historically by van Doorn et al. [21] and recently in Ground Zero [60].

Modeling *other formal systems* may also be of interest. Xu and Maillard [71] formalized the soundness of presheaf semantics for geometric logic in Lean. Mathlib already contains definitions of first-order theories and structures, such as rings and graphs. Unfortunately, constructing first-order sentences or proofs by hand suffers from the kinds of issues described in Section 5. Applying `SynthLean`-like proof assistance would simplify working internally in these theories.

One may also view `SynthLean` theories as *embedded domain-specific languages*. From this perspective, we find an analogy with an interpretation of the simply typed λ-calculus in cartesian closed categories due to Elliott [35]. This work, done in Haskell, had no formally established properties.

The idea of *a proof assistant for deeply embedded MLTT* has been proposed before. To our best knowledge this has never materialized, though some of the aforementioned certification projects could likely provide it with marginal effort. Bickford [17] reported progress towards an assistant for cubical type theory embedded in Nuprl.

7 Conclusion

We have presented `SynthLean`, a proof assistant that unifies reasoning in Martin-Löf type theories with reasoning about their models. On the syntactic side, we developed a concrete grammar for theories with extensional Π and Σ types, intensional identity types, and finitely many universes, together with a lightweight metatheory supporting substitution, inversion, and uniqueness of typing. On the semantic side, we formalized natural model semantics in presheaf categories, and proved a strict soundness theorem guaranteeing that syntactic derivations admit well-behaved interpretations in any model. On top of these components, we implemented a frontend that allows Lean users to construct objects and proofs both internally in a type theory and externally in its semantics, with a typechecker that certifies internal constructions.

In future work, we seek to simplify and generalize the categorical semantics of MLTT. Work towards generalized natural model semantics includes Martin-Löf algebras [12], the theory of clans [45], and categories with representable maps [68]. As a concrete application of `SynthLean`, we also aim to complete a construction of the groupoid model of HoTT0 and develop mathematics internally to that model. This could be used to define and compute cohomology groups via Eilenberg-MacLane spaces [51]. Finally, although reusing Lean's native elaboration is very convenient, it restricts `SynthLean` to theories with the same equational and judgmental structure, excluding others such as extensional or cubical type theory. Developing more flexible elaboration strategies is therefore a potential goal.

Acknowledgements

The authors wish to thank Gabriel Ebner at Microsoft Research for advising an internship during which the first author developed a prototype of the frontend, Aaron Liu for contributing the `rw!` tactic to Mathlib, Michael Shulman for suggesting a focus on theories, Jeremy Avigad, Jasmin Blanchette, and Jan Johannsen for comments on a draft, and the anonymous reviewers for improvement suggestions.

This material is based upon work supported by the Air Force Office of Scientific Research under MURI award FA9550-21-1-0009, as well as by the National Science Foundation under grant number 2434614. Any opinions, findings, or recommendations expressed here are those of the authors and do not necessarily reflect the views of the NSF. One author, Yiming Xu, is co-funded by the European Union (ERC, Nekoka, 101083038). Views and opinions expressed are however those of the authors only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

References

[1] Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. 1991. Explicit Substitutions. *J. Funct. Program.* 1, 4 (1991), 375–416. doi:10.1017/S0956796800000186

[2] Michael Abbott, Thorsten Altenkirch, and Neil Ghani. 2003. Categories of Containers. In *Foundations of Software Science and Computation Structures*, Andrew D. Gordon (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 23–38. doi:10.1007/3-540-36576-1_2

[3] Andreas Abel. 2013. *Normalization by evaluation: Dependent types and impredicativity.* Ph. D. Dissertation. Ludwig-Maximilians-Universität München.

[4] Andreas Abel, Joakim Öhman, and Andrea Vezzosi. 2017. Decidability of conversion for type theory in type theory. *Proc. ACM Program. Lang.* 2, POPL, Article 23 (Dec. 2017), 29 pages. doi:10.1145/3158111

[5] C. B. Aberlé and David I. Spivak. 2025. Polynomial Universes in Homotopy Type Theory. arXiv:2409.19176 [cs.LO] https://arxiv.org/abs/2409.19176

[6] Arthur Adjedj, Meven Lennon-Bertrand, Kenji Maillard, Pierre-Marie Pédrot, and Loïc Pujet. 2024. Martin-Löf à la Coq. In *Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs* (London, UK) *(CPP 2024)*. Association for Computing Machinery, New York, NY, USA, 230–245. doi:10.1145/3636501.3636951

[7] Benedikt Ahrens, Peter LeFanu Lumsdaine, and Vladimir Voevodsky. 2017. Categorical Structures for Type Theory in Univalent Foundations. In *26th EACSL Annual Conference on Computer Science Logic (CSL 2017) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 82)*, Valentin Goranko and Mads Dam (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 8:1–8:16. doi:10.4230/LIPIcs.CSL.2017.8

[8] Benedikt Ahrens, Paige Randall North, and Niels van der Weide. 2023. Bicategorical type theory: semantics and syntax. *Mathematical Structures in Computer Science* 33, 10 (2023), 868–912. doi:10.1017/S0960129523000312

[9] Thorsten Altenkirch and Ambrus Kaposi. 2016. Type theory in type theory using quotient inductive types. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. Association for Computing Machinery, New York, NY, USA, 18–29. doi:10.1145/2837614.2837638

[10] Carlo Angiuli and Daniel Gratzer. 2024. *Principles of Dependent Type Theory.* https://www.danielgratzer.com/papers/type-theory-book.pdf

[11] Steve Awodey. 2018. Natural models of homotopy type theory. *Mathematical Structures in Computer Science* 28, 2 (2018), 241–286. doi:10.1017/S0960129516000268

[12] Steve Awodey. 2025. Algebraic Type Theory, Part 1: Martin-Löf algebras. *arXiv preprint arXiv:2505.10761v1* (2025).

[13] Andrej Bauer. 2006. First Steps in Synthetic Computability Theory. *Electronic Notes in Theoretical Computer Science* 155 (2006), 5–31. doi:10.1016/j.entcs.2005.11.049 Proceedings of the 21st Annual Conference on Mathematical Foundations of Programming Semantics (MFPS XXI).

[14] Andrej Bauer, Jason Gross, Peter LeFanu Lumsdaine, Michael Shulman, Matthieu Sozeau, and Bas Spitters. 2017. The HoTT library: a formalization of homotopy type theory in Coq. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs* (Paris, France) *(CPP 2017)*. Association for Computing Machinery, New York, NY, USA, 164–172. doi:10.1145/3018610.3018615

[15] Andrej Bauer, Philipp Haselwarter, and Théo Winterhalter. 2017. A modular formalization of type theory in Coq. *TYPES 2017* (2017). https://types2017.elte.hu/proc.pdf#page=39

[16] Marc Bezem, Thierry Coquand, and Simon Huber. 2014. A Model of Type Theory in Cubical Sets. In *19th International Conference on Types for Proofs and Programs (TYPES 2013) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 26)*, Ralph Matthes and Aleksy Schubert (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 107–128. doi:10.4230/LIPIcs.TYPES.2013.107

[17] Mark Bickford. 2020. Cubical type theory with several universes in Nuprl. https://hott-uf.github.io/2020/HoTTUF_2020_paper_12.pdf

[18] Ingo Blechschmidt. 2021. Using the internal language of toposes in algebraic geometry. *arXiv preprint arXiv:2111.03685* (2021).

[19] Simon Boulier, Pierre-Marie Pédrot, and Nicolas Tabareau. 2017. The next 700 syntactical models of type theory. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs* (Paris, France) *(CPP 2017)*. Association for Computing Machinery, New York, NY, USA, 182–194. doi:10.1145/3018610.3018620

[20] Ana Bove and Venanzio Capretta. 2005. Modelling general recursion in type theory. *Math. Struct. Comput. Sci.* 15, 4 (2005), 671–708. doi:10.1017/S0960129505004822

[21] Ulrik Buchholtz, Floris van Doorn, and Jakob von Raumer. 2017. Homotopy Type Theory in Lean. *CoRR* abs/1704.06781 (2017). arXiv:1704.06781 http://arxiv.org/abs/1704.06781

[22] Mario Carneiro. 2025. Lean4Lean: Verifying a Typechecker for Lean, in Lean. arXiv:2403.14064 [cs.PL] https://arxiv.org/abs/2403.14064

[23] John Cartmell. 1986. Generalised algebraic theories and contextual categories. *Annals of Pure and Applied Logic* 32 (1986), 209–243. doi:10.1016/0168-0072(86)90053-9

[24] Simon Castellan, Pierre Clairambault, and Peter Dybjer. 2015. Undecidability of Equality in the Free Locally Cartesian Closed Category. In *13th International Conference on Typed Lambda Calculi and Applications (TLCA 2015) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 38)*, Thorsten Altenkirch (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 138–152. doi:10.4230/LIPIcs.TLCA.2015.138

[25] James Chapman, Pierre-Évariste Dagand, Conor McBride, and Peter Morris. 2010. The gentle art of levitation. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming* (Baltimore, Maryland, USA) *(ICFP '10)*. Association for Computing Machinery, New York, NY, USA, 3–14. doi:10.1145/1863543.1863547

[26] Felix Cherubini, Thierry Coquand, Freek Geerligs, and Hugo Moeneclaey. 2024. A Foundation for Synthetic Stone Duality. In *30th International Conference on Types for Proofs and Programs, TYPES 2024, June 10-14, 2024* (Copenhagen, Denmark) *(LIPIcs, Vol. 336)*, Rasmus Ejlers Møgelberg and Benno van den Berg (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 3:1–3:20. doi:10.4230/LIPICS.TYPES.2024.3

[27] The Mathlib Community. 2020. The Lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs* (New Orleans, LA, USA) *(CPP 2020)*. Association for Computing Machinery, New York, NY, USA, 367–381. doi:10.1145/3372885.3373824

[28] Thierry Coquand. 2013. Presheaf model of type theory. (2013).

[29] P.-L. Curien. 1991. An abstract framework for environment machines. *Theoretical Computer Science* 82, 2 (1991), 389–402. doi:10.1016/0304-3975(91)90230-Y

[30] Menno de Boer. 2020. *A Proof and Formalization of the Initiality Conjecture of Dependent Type Theory.* Licentiate Thesis. Stockholm University, Department of Mathematics.

[31] Kunhong Du. 2025. *On the Formalization of the Simplicial Model of HoTT.* Master's thesis. Rheinischen Friedrich-Wilhelms-Universität Bonn.

[32] Peter Dybjer. 1995. Internal Type Theory. In *Types for Proofs and Programs, International Workshop TYPES'95, June 5-8, 1995, Selected Papers* (Torino, Italy) *(Lecture Notes in Computer Science, Vol. 1158)*, Stefano Berardi and Mario Coppo (Eds.). Springer, 120–134. doi:10.1007/3-540-61780-9_66

[33] Peter Dybjer. 2000. A General Formulation of Simultaneous Inductive-Recursive Definitions in Type Theory. *J. Symb. Log.* 65, 2 (2000), 525–549. doi:10.2307/2586554

[34] Gabriel Ebner. 2022. quote4: Intuitive, type-safe expression quotations for Lean 4. https://github.com/leanprover-community/quote4

[35] Conal Elliott. 2017. Compiling to categories. *Proc. ACM Program. Lang.* 1, ICFP, Article 27 (Aug. 2017), 27 pages. doi:10.1145/3110271

[36] Andrei P. Ershov. 1958. On Programming of Arithmetic Operations. *Commun. ACM* 1, 8 (1958), 3–9. doi:10.1145/368892.368907

[37] Nicola Gambino and Joachim Kock. 2013. Polynomial functors and polynomial monads. *Mathematical Proceedings of the Cambridge Philosophical Society* 154, 1 (2013), 153–192. doi:10.1017/S0305004112000394

[38] Daniel R. Grayson, Benedikt Ahrens, Ralph Matthes, Niels van der Weide, Anders Mörtberg, cathlelay, Langston Barrett, Peter LeFanu Lumsdaine, Marco Maggesi, Vladimir Voevodsky, Tomi Pannila, Gianluca Amato, Michael Lindgren, Mitchell Riley, Skantz, CalosciMatteo, Arnoud van der Leer, Tobias-Schmude, Daniel Frumin, Auke Booij, Hichem Saghrouni, Dennis, varkor, Dimitris Tsementzis, niccoloveltri, Anthony Bordg, Dominik Kirst, Karl Palmskog, Satoshi Kura, and Tamara von Glehn. 2025. *UniMath/UniMath: v20250923*. doi:10.5281/zenodo.17186647

[39] Sina Hazratpour. 2024. Poly: A Lean4 Formalization of Polynomial Functors. https://github.com/sinhp/poly

[40] Martin Hofmann. 1995. On the interpretation of type theory in locally cartesian closed categories. In *Computer Science Logic*, Leszek Pacholski and Jerzy Tiuryn (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 427–441. doi:10.1007/BFb0022273

[41] Martin Hofmann and Thomas Streicher. 1998. The Groupoid Interpretation of Type Theory. In *Twenty-five years of constructive type theory (Venice, 1995)*. Oxford Logic Guides, Vol. 36. Oxford Univ. Press, New York, 83–111. doi:10.1093/oso/9780198501275.003.0008

[42] Joseph Hua, Steve Awodey, Mario Carneiro, Sina Hazratpour, Wojciech Nawrocki, Spencer Woolfson, and Yiming Xu. 2025. HoTTLean: Formalizing the Meta-Theory of HoTT in Lean. *TYPES 2025* (2025).

[43] J.M.E. Hyland. 2006. First steps in synthetic domain theory. In *Category Theory: Proceedings of the International Conference held in Como, Italy, July 22–28, 1990*. Springer, 131–156. doi:10.1007/BFb0084217

[44] Bart Jacobs. 1999. *Categorical Logic and Type Theory.* Number 141 in Studies in Logic and the Foundations of Mathematics. North Holland, Amsterdam. doi:10.1016/S0049-237X(98)X8028-6

[45] André Joyal. 2017. Notes on clans and tribes. *arXiv preprint arXiv:1710.10238* (2017).

[46] Ambrus Kaposi and Loïc Pujet. 2025. Type Theory in Type Theory using a Strictified Syntax. *Proc. ACM Program. Lang.* 9, ICFP, Article 266 (Aug. 2025), 31 pages. doi:10.1145/3747535

[47] Chris Kapulkin and Peter LeFanu Lumsdaine. 2018. The Simplicial Model of Univalent Foundations (after Voevodsky). arXiv:1211.2851 [math.LO] https://arxiv.org/abs/1211.2851

[48] Anders Kock. 2006. *Synthetic Differential Geometry* (2 ed.). Cambridge University Press. doi:10.1017/CBO9780511550812

[49] András Kovács. 2022. Generalized Universe Hierarchies and First-Class Universe Levels. In *30th EACSL Annual Conference on Computer Science Logic, CSL 2022, February 14-19, 2022, Göttingen, Germany (Virtual Conference) (LIPIcs, Vol. 216)*, Florin Manea and Alex Simpson (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 28:1–28:17. doi:10.4230/LIPICS.CSL.2022.28

[50] Meven Lennon-Bertrand. 2025. What Does It Take to Certify a Conversion Checker?. In *10th International Conference on Formal Structures for Computation and Deduction, FSCD 2025, July 14-20, 2025, Birmingham, UK (LIPIcs, Vol. 337)*, Maribel Fernández (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 27:1–27:23. doi:10.4230/LIPICS.FSCD.2025.27

[51] Daniel R. Licata and Eric Finster. 2014. Eilenberg-MacLane spaces in homotopy type theory. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)* (Vienna, Austria) *(CSL-LICS '14)*. Association for Computing Machinery, New York, NY, USA, Article 66, 9 pages. doi:10.1145/2603088.2603153

[52] Daniel R. Licata and Michael Shulman. 2013. Calculating the Fundamental Group of the Circle in Homotopy Type Theory. In *28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, New Orleans, LA, USA, June 25-28, 2013*. IEEE Computer Society, 223–232. doi:10.1109/LICS.2013.28

[53] Bohdan Liesnikov and Jesper Cockx. 2024. Building a Correct-by-Construction Type Checker for a Dependently Typed Core Language. In *Programming Languages and Systems: 22nd Asian Symposium, APLAS 2024, Kyoto, Japan, October 22-24, 2024, Proceedings* (Kyoto, Japan). Springer-Verlag, Berlin, Heidelberg, 63–83. doi:10.1007/978-981-97-8943-6_4

[54] Ieke Moerdijk and Erik Palmgren. 2000. Wellfounded trees in categories. *Annals of Pure and Applied Logic* 104, 1 (2000), 189–218. doi:10.1016/S0168-0072(00)00012-9

[55] Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction – CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings*. Springer-Verlag, Berlin, Heidelberg, 625–635. doi:10.1007/978-3-030-79876-5_37

[56] Wojciech Nawrocki, Joseph Hua, Mario Carneiro, Yiming Xu, Spencer Woolfson, Shuge Rong, Sina Hazratpour, and Steve Awodey. 2025. *Implementation of "A Certifying Proof Assistant for Synthetic Mathematics in Lean"*. doi:10.5281/zenodo.17798253

[57] George C. Necula and Peter Lee. 1998. The design and implementation of a certifying compiler. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation* (Montreal, Quebec, Canada) *(PLDI '98)*. Association for Computing Machinery, New York, NY, USA, 333–344. doi:10.1145/277650.277752

[58] Michael Rathjen. 2017. Proof Theory of Constructive Systems: Inductive Types and Univalence. In *Feferman on Foundations: Logic, Mathematics, Philosophy*, Gerhard Jäger and Wilfried Sieg (Eds.). Springer International Publishing, Cham, 385–419. doi:10.1007/978-3-319-63334-3_14

[59] Egbert Rijke, Elisabeth Stenholm, Jonathan Prieto-Cubides, Fredrik Bakke, Vojtěch Štěpančík, and others. 2025. *The Agda-Unimath library*. https://github.com/UniMath/agda-unimath/

[60] rzrn. 2025. *Ground Zero: Lean 4 HoTT Library*. https://github.com/rzrn/ground_zero

[61] Steven Schäfer, Tobias Tebbi, and Gert Smolka. 2015. Autosubst: Reasoning with De Bruijn Terms and Parallel Substitutions. In *Interactive Theorem Proving - 6th International Conference, ITP 2015, August 24-27, 2015, Proceedings* (Nanjing, China) *(Lecture Notes in Computer Science, Vol. 9236)*, Christian Urban and Xingyuan Zhang (Eds.). Springer, 359–374. doi:10.1007/978-3-319-22102-1_24

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

[63] Matthieu Sozeau and Nicolas Tabareau. 2014. Towards an internalization of the groupoid model of type theory. *Types for Proofs and Programs 20th Meeting (TYPES 2014), Book of Abstracts* (2014).

[64] Jonathan Sterling. 2021. *First Steps in Synthetic Tait Computability*. Ph. D. Dissertation. Carnegie Mellon University.

[65] Thomas Streicher. 1991. *Semantics of type theory - correctness, completeness and independence results*. Birkhäuser. doi:10.1007/978-1-4612-0433-6

[66] Pierre-Yves Strub, Nikhil Swamy, Cedric Fournet, and Juan Chen. 2012. Self-certification: bootstrapping certified typecheckers in F* with Coq. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Philadelphia, PA, USA) *(POPL '12)*. Association for Computing Machinery, New York, NY, USA, 571–584. doi:10.1145/2103656.2103723

[67] The Agda Community. 2025. *Cubical Agda Library*. https://github.com/agda/cubical

[68] Taichi Uemura. 2023. A general framework for the semantics of type theory. *Mathematical Structures in Computer Science* 33, 3 (2023), 134–179. doi:10.1017/S0960129523000208

[69] Sebastian Ullrich and Leonardo de Moura. 2022. Beyond Notations: Hygienic Macro Expansion for Theorem Proving Languages. *Log. Methods Comput. Sci.* 18, 2 (2022). doi:10.46298/LMCS-18(2:1)2022

[70] Paweł Wieczorek and Dariusz Biernacki. 2018. A Coq formalization of normalization by evaluation for Martin-Löf type theory. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPPa2018, Los Angeles, CA, USA, January 8-9, 2018*, June Andronick and Amy P. Felty (Eds.). ACM, 266–279. doi:10.1145/3167091

[71] Yiming Xu and Kenji Maillard. 2025. Geometric Reasoning in Lean: from Algebraic Structures to Presheaves. In *31st International Conference on Types for Proofs and Programs*.

Received 2025-09-13; accepted 2025-11-13