ZFLean: a framework for set-level mathematics in Lean

Vincent Trélat
Université de Lorraine, CNRS, INRIA, Nancy, France

Abstract

We present ZFLean, a Lean 4 library for doing core mathematics inside a model of ZFC with the ergonomics expected of typed Mathlib developments. Building on Mathlib's ZFC model, we contribute a relational calculus for sets with rewriting hints and small predictable tactics, canonical set-theoretic constructions—Booleans, naturals, integers, sums/option—and bridges between ZFC objects and Lean's native types enabling mixed set-level/typed proofs. The layer reduces boilerplate for extensional reasoning while remaining compatible with vanilla Mathlib. We discuss library organization and usage patterns that lower the friction of set-theoretic formalization in a dependently typed assistant. We demonstrate typical use of the framework with a case study exercising our constructions and relational calculus through a proof of an isomorphism theorem on curried functions.

2012 ACM Subject Classification Theory of computation → Logic and verification

Keywords and phrases Set theory, ZFC, Lean, Theorem proving

Digital Object Identifier 10.4230/LIPIcs...

Funding *Vincent Trélat*: ANR project BLaSST (ANR-21-CE25-0010).

1 Introduction

Modern mathematical formalization is increasingly carried out with proof assistants, whose foundational cores range from first-order set theories to more expressive systems based on simple or dependent type theory. This evolution was fueled by the *proofs-as-objects* vision of the Curry-Howard correspondence, and it crystallized into two broad and often contrasted families of foundations: *collections*-based theories (set-theoretic) and *constructions*-based theories (type-theoretic). Various proposals aim to reconcile these viewpoints, either by generalizing over them—e.g. category theory—or by refining them—e.g. constructive set theories such as CZF [2].

Nevertheless, formalizing *set-level* mathematics is sometimes desirable—for extensional equalities, partiality via relations, or category-style "up to isomorphism" reasoning. However, working directly at set level in a typed assistant can be verbose and brittle: partial maps require encodings; extensionality must be disciplined; and crossing the boundary to native types is manual and error-prone. As a result, ZFC models are more often referenced than *used* for development. We therefore leverage Lean's Mathlib [11] and its provided model of ZFC and develop a framework for doing mathematics in ZFC with Lean-grade ergonomics. Our aim is *not* to replace Mathlib's native arithmetic or data types; instead, we engineer a uniform *relational calculus* (with partial automation), provide *canonical ZFC constructions* with *universal properties*, and build *bridges* to Lean 4 types, so proofs can fluidly toggle between extensional sets and typed infrastructure. We contribute the following:

- *a relational calculus*: Composition/converse/images with associativity/identity and image, composition interaction laws, normal forms, and rewriting hints feeding partial automation.
- *canonical constructions with usual properties*: $\mathbb{B}$, $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, functions, sums, options.
- *an interoperable framework with Lean 4/Mathlib*: Bridges allowing to step across boundaries smoothly, enabling "up to isomorphism" reasoning and transport of properties.

All definitions and theorems mentioned in this paper are available in the artifacts, as well as full proofs omitted in the text.

We first recall Mathlib's ZFC model and design choices in Section 2. In Section 3 we present the relational calculus with automation tactics for relations and (partial) functions. Using this calculus, we build canonical constructions along with their usual associated properties in Section 4. In Section 4.2 we formalize embeddings and isomorphisms, along with structure theorems and transport patterns. Section 5 then demonstrates usability of the framework via selected case studies. We also discuss related work in Section 6 before concluding.

2 Background and Design Choices

We rely on Mathlib's implementation of a model of ZFC set theory [6], which we briefly recall here. The construction starts from a universe-polymorphic type of *pre-sets* `PSet` and its extensional quotient, so as to obtain the intended set-theoretic model. We briefly recall central definitions: extensional equivalence $X \sim Y$ and membership $x \in X$, which witness the axioms of extensionality and regularity in the model.

2.1 Pre-sets

Pre-sets are defined inductively as universe-polymorphic structures, as follows.

▶ **Definition 2.1.** *For any type α in a universe of level u*[1] *and any indexed family A of pre-sets over α, a pre-set can be constructed from α and A. This provides an inductive (intensional) definition of pre-sets via a constructor of type $\prod_{\alpha\,:\,\mathsf{Type}\,u}(\alpha \to \mathsf{PSet}) \to \mathsf{PSet}$. The corresponding Lean definition is shown below.*

```
inductive PSet : Type (u + 1)
  | mk (α : Type u) (A : α → PSet) : PSet
```

Constructing a pre-set is therefore a cumulative process, a commonly observed fact about sets: (pre-)sets are built from other (pre-)sets. With such a definition, one can inductively construct a pre-set upon an underlying type α by providing an interpretation function of type $\alpha \to \mathsf{PSet}$ of its elements as pre-sets. Since we need a starting point to populate the type of pre-sets, we construct the empty pre-set. This shows that the type of pre-sets is *inhabited*.

▶ **Definition 2.2.** *The empty pre-set is obtained from the (universe polymorphic) empty type $\bot$ and its elimination function $\mathsf{elim}_\bot : \prod_\alpha \bot \to \alpha$*

$$\varnothing \triangleq \mathsf{PSet.mk}\ \bot\ \mathsf{elim}_\bot$$

Operations on pre-sets are then defined with the aim of being later lifted to sets. It is noticeable that some of these operations will serve as witnesses of the ZFC axioms. We only recall principal operations here, such as extensional equivalence and membership.

[1] $\mathsf{Type}\,u$ denotes Lean's universe level u in a cumulative hierarchy where the parameter u makes the construction universe-polymorphic.

▶ **Definition 2.3.** *Two pre-sets* $X =: \langle \alpha, A \rangle$ *and* $Y =: \langle \beta, B \rangle$ *are said to be extensionally equivalent—or to have the same extension—denoted by* $X \sim Y$*, if every element of* A *is (inductively) extensionally equivalent to some element of* B *and vice-versa:*

$$X \sim Y \triangleq (\forall\, a,\, \exists\, b,\, A(a) \sim B(b)) \wedge (\forall\, b,\, \exists\, a,\, A(a) \sim B(b))$$

Extensional equivalence is then shown to be an equivalence relation on pre-sets, which allows us to instantiate a setoid structure on pre-sets and serves as a witness for the axiom of extensionality. It is also used to define pre-set membership, as follows.

▶ **Definition 2.4.** *A pre-set* x *is said to be a member of a pre-set* $X =: \langle \alpha, A \rangle$*, denoted by* $x \in X$*, if it is extensionally equivalent to some element of* X*:*

$$x \in X \triangleq \exists\, a,\, x \sim A(a)$$

The membership relation is then shown to be well-founded—meaning there is no infinite $\in$-chain of pre-sets—a key property that witnesses the axiom of regularity. Further usual operations such as cartesian product ($\times$), union ($\cup$), and powerset ($\mathcal{P}$), are defined on pre-sets, all to be later lifted to sets.

2.2 Sets

Sets are defined as the extensional quotient of pre-sets, as follows.

▶ **Definition 2.5.** *The type of sets is defined as the quotient type* $\mathsf{ZFSet} \triangleq \mathsf{PSet}/\sim$.

Since Lean has built-in support for quotient types, the definition of sets in Lean is straightforward. Operations on sets are then defined by lifting the corresponding operations on pre-sets via the quotient map.

▶ Remark 2.6. The construction validates the full ZFC axioms, including choice: in ZFLean we only rely on choice to define noncomputable selectors such as function evaluation on partial functions. This definition already marks the divergence between *intensional* and *extensional* theories. ZFC sets are indeed constrained by a notion of equality that is coarser than *definitional* equality, which is fundamentally used in many systems, among which Lean, Rocq and Agda. Indeed, although most proof assistants are based on intensional type systems, some like F* [10] treat provably equal terms like definitionally equal ones. The purpose of our development is exactly to alleviate the friction caused by this divergence by preventing repeated quotient-lifting boilerplate: proof scripts stay at the `ZFSet` interface, and *extensional* reasoning inside Lean is enabled while retaining compatibility with its *intensional* core.

A definition of Kuratowski's ordered pairs is also provided by the model, denoted by (x, y) for any sets x and y and is defined as the set $\{\{x\}, \{x, y\}\}$, but lacks projections and simplification lemmas. We therefore define projections π_1 and π_2 accordingly and provide simplification lemmas to manipulate them. This represents the first contribution of our work.

▶ **Definition 2.7.** *We define general projections* π_1 *and* π_2 *for any set* x*:*

$$\pi_1\, x \triangleq \bigcup\bigcap x \quad \textit{and} \quad \pi_2\, x \triangleq \begin{cases} \pi_1\, x & \textit{if } \bigcup x \setminus \bigcap x = \varnothing \\ \bigcup (\bigcup x \setminus \bigcap x) & \textit{otherwise} \end{cases}$$

▶ Remark 2.8. The projections π_1 and π_2 are defined as *total* set-level operators to avoid partiality in the object language; they are only meant to be used under hypotheses that force x to be a pair, e.g. it belongs to a cartesian product $x \in A \times B$, or $x = (a, b)$. When x is *not* a Kuratowski pair, $\pi_1(x)$ and $\pi_2(x)$ are just some sets with no intended semantic content.

Listing 1 Projections for Kuratowski pairs along with simplification lemmas.

```
theorem π₁_pair (x y : ZFSet) : π₁ (x.pair y) = x := ⋯
theorem π₂_pair (x y : ZFSet) : π₂ (x.pair y) = y := ⋯
theorem pair_eta {z A B : ZFSet} (h : z ∈ A × B) :
  z = z.π₁.pair z.π₂ := ⋯
theorem pair_mem_prod {x y a b : ZFSet} :
  a.pair b ∈ x × y ↔ a ∈ x ∧ b ∈ y := ⋯
```

With these projections, we can prove the expected simplification lemmas for ordered pairs, e.g. $\pi_1(x, y) = x$ and $\pi_2(x, y) = y$ for any sets x and y, η-expansion of pairs and membership on a cartesian product, as shown in Listing 1—with proofs omitted and replaced by ⋯ but available in the artifacts, with additional equational lemmas.

ZFC set theory has no built-in notion of functions, whose definition must be taken carefully. In programming languages and type theories, functions are usually first-class and *total* by construction, while mathematical functions can be *partial* and are defined as *functional relations* whose domain may be smaller than the intended source set. This difference is usually bridged by using options or dependent types to encode partiality, however in this work we aim to stay as close as possible to the set-theoretic definitions. Also, Mathlib already provides a definition for total functions and exponential objects, which we reuse in our development. The definition is as follows.

▶ **Definition 2.9.** *A set f is said to be a total function from a set A to a set B, denoted by* $\mathsf{IsFunc}(A, B, f)$*, if it is a functional relation whose domain is exactly A:*

$$\mathsf{IsFunc}(A, B, f) \triangleq f \subseteq A \times B \wedge \forall x \in A, \exists! y \in B, (x, y) \in f$$

The exponential object B^A is then defined as the set of all functions from A to B:

$$B^A \triangleq \{f \in \mathcal{P}(A \times B) \mid \mathsf{IsFunc}(A, B, f)\}$$

This definition, however, does not cover partial functions, which are ubiquitous in mathematical developments. This highly motivates the relational calculus presented in the next section.

3 A Relational Calculus in ZFC

Throughout this section we work inside the ZFC model provided by Mathlib and extend it with a calculus for binary relations and partial/total functions. A *binary relation* R between A and B is a subset $R \subseteq A \times B$. Our calculus packages standard operations on relations (converse, identity, composition, domain, range, image) together with predicates for partial and total functions, and supplies small *automation tactics* that try to discharge ubiquitous well-formedness side-conditions, so that these objects can be manipulated *as is* without cumbersome boilerplate, as on paper. Those tactics are called `zrel`, `zpfun`, and `zfun`, and will be described later.

3.1 Basic definitions

▶ **Definition 3.1** (converse, identity, composition)**.** *Let $R \subseteq A \times B$ and $S \subseteq B \times C$ be relations.*

- *The* identity relation $\mathbb{1}_A$ *on* A *is defined as:*

$$\mathbb{1}_A \triangleq \{x \in A \times A \mid \pi_1(x) = \pi_2(x)\}$$

The identity relation is shown to be a total function belonging to A^A*, and later shown to be a bijection and the neutral element for composition.*

- *The* converse R^{-1} *is defined as:*

$$R^{-1} \triangleq \{z \in B \times A \mid \exists\, x\, y,\ x \in A \wedge y \in B \wedge z = (y, x) \wedge (x, y) \in R\}$$

This definition depends on the side-condition $R \subseteq A \times B$ *(declaring that* R *is a relation between the implicit sets), an implicit argument that* `zrel` *attempts to discharge in ZFLean.*

- *The* composition $S \circ R \subseteq A \times C$ *is defined as:*

$$S \circ R := \{w \in A \times C \mid \exists\, x\, z,\ x \in A \wedge z \in C \wedge w = (x, z) \wedge \exists\, y \in B,\ (x, y) \in R \wedge (y, z) \in S\}$$

Unlike the previous definition, nothing is enforced on R *and* S*: if either* R *or* S *is not a relation, the composition is still well-defined, but is empty. However, we define* functional composition $\circ^{\mathsf{z}}$ *specifically for functions, requiring* R *and* S *to be functions, and using the* `zfun` *tactic to try to discharge the side-conditions automatically.*

▶ **Definition 3.2** (Domain, range, image)**.** *For a relation* $R \subseteq A \times B$*, its* domain, range, *and* image *of a set* $X \subseteq A$ *are defined as:*

$$\begin{aligned} \mathsf{Dom}(R) &\triangleq \{x \in A \mid \exists\, y \in B,\ (x, y) \in R\} \\ \mathsf{Range}(R) &\triangleq \{y \in B \mid \exists\, x \in \mathsf{Dom}(R),\ (x, y) \in R\} \\ R[X] &\triangleq \{y \in B \mid \exists\, x \in X,\ (x, y) \in R\} \end{aligned}$$

Operationally, all three definitions above invoke the `zrel` *tactic to discharge the side-condition that* R *is a relation between* A *and* B*.*

Finally, we define partial functions as functional relations without totality requirements.

▶ **Definition 3.3** (Partial functions)**.** *A set* f *is said to be a partial function from set* A *to set* B*, written* $\mathsf{IsPFunc}(f, A, B)$*, if it is* functional*:*[2]

$$\mathsf{IsPFunc}(f, A, B) \triangleq f \subseteq A \times B \wedge (\forall\, x \in A,\ \forall\, y \in B,\ \forall\, z \in B,\ (x, y) \in f \wedge (x, z) \in f \Rightarrow y = z)$$

We also derive predicates IsInjective, IsSurjective, and IsBijective in the usual sense, which all invoke the tactic `zfun` to discharge functional totality requirements.

3.2 Function evaluation and λ-abstraction

We introduce two convenient features for working with functions, namely function application and λ-abstraction, defined as follows. We strive both to keep standard mathematical notation and to automate side-conditions as much as possible.

- *function application* that maps $x \in \mathsf{Dom}(f)$ to the unique y such that $(x, y) \in f$ and denoted by $@^{\mathsf{z}} f(x)$, invoking the `zpfun` tactic; we show that evaluation commutes with composition in the expected way: $@^{\mathsf{z}}(g \circ^{\mathsf{z}} f)(x) = @^{\mathsf{z}} g(@^{\mathsf{z}} f(x))$, and $@^{\mathsf{z}} f(x)$ coincides with the unique element of the image $f[\{x\}]$. In practice, $@^{\mathsf{z}}$ is a non-computable dependently

[2] The order of arguments is voluntarily changed compared to IsFunc, so that in Lean, one may write `f.IsPFunc A B`

typed operator that requires a proof that f is a partial function and a proof h_x that x belongs to its domain (that must be provided, there is no automation at this point), denoted $@^z f(\langle x, h_x\rangle)$. Its definition makes use of the axiom of choice to select an element y such that $(x, y) \in f$. Since f is functional, there is at most one such y, and since x belongs to the domain of f, there is at least one such y, hence such a y exists and is unique.[3]

- λ-abstraction constructing functions from expressions, written as:

$$\begin{array}{lcccc} \lambda^z & : & A & \to & B \\ & | & x & \mapsto & e(x) \end{array} \qquad \text{is the relation} \qquad \{(x, y) \in A \times B \mid y = e(x)\},$$

which is then shown to belong to B^A whenever $e(x) \in B$ for all $x \in A$. We prove the expected equational properties for this notation, summarized in the following theorem.

▶ **Theorem 3.4** (Specification, extensionality, β- and η-conversion). *Let A, B be sets, $f \in B^A$, and e, e' be unary Lean-level (endo)functions on sets.*

- ***Specification:*** *For any sets x and y, the following holds:*

$$(x, y) \in \begin{array}{lcccc} \lambda^z & : & A & \to & B \\ & | & x & \mapsto & e(x) \end{array} \iff x \in A \wedge y \in B \wedge y = e(x)$$

- ***Extensionality:***

$$\left(\begin{array}{lcccc} \lambda^z & : & A & \to & B \\ & | & x & \mapsto & e(x) \end{array}\right) = \left(\begin{array}{lcccc} \lambda^z & : & A & \to & B \\ & | & x & \mapsto & e'(x) \end{array}\right) \iff \forall x \in A,\, e(x) = e'(x)$$

- ***β-reduction:***

$$\forall x \in A,\, @^z \left(\begin{array}{lcccc} \lambda^z & : & A & \to & B \\ & | & x & \mapsto & e(x) \end{array}\right) (\langle x, \cdots\rangle) = e(x)$$

- ***η-expansion:***

$$f = \left(\begin{array}{lcccc} \lambda^z & : & A & \to & B \\ & | & x & \mapsto & @^z f(\langle x, \cdots\rangle) \end{array}\right)$$

Proof. Proofs are carried out by applying extensionality on the corresponding sets. Full proof details are available in the artifacts as proofs of the theorems `lambda_spec`, `lambda_ext_iff`, `fapply_lambda`, and `lambda_eta`. ◀

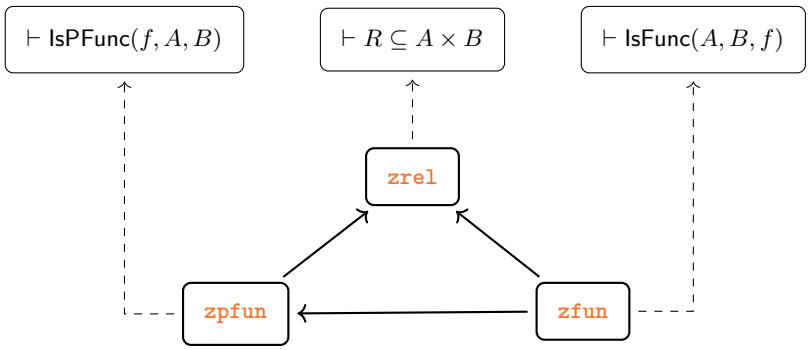
Application and abstraction integrate well with rewriting, so that many proofs reduce to algebra on images, evaluation, and composition, as expected. We illustrate this with a simple example below.

▶ **Example 3.5.** Consider the extensionality principle for total functions: two functions $f, g \in B^A$ are equal if they are pointwise equal on A. In ZFLean, this is stated as the following theorem:

```
theorem is_func_ext_iff {A B : ZFSet} {f g : ZFSet}
  (hf : IsFunc A B f) (hg : IsFunc A B g) :
    f = g ↔ ∀ x ∈ A, @ᶻf ⟨x, ⋯⟩ = @ᶻg ⟨x, ⋯⟩ := ⋯
```

[3] This property is proved in ZFLean as `IsPFunc.exists_unique_of_mem_dom`.

Figure 1 Overview of the automation tactics for relations and functions. Dashed arrows indicate the shape of the goals discharged by each tactic; thick arrows indicate tactic invocations.



where the first two omitted proofs (⋯) are proofs that x belongs to the domain of f and g respectively. The proof of this theorem can be carried out directly using extensionality on sets, although it turns out to be rather tedious. It is in fact easier to η-expand both f and g and apply extensionality from Theorem 3.4 on the resulting λ-abstractions. In ZFLean, this amounts to the following rewriting steps:

```
-- hf : IsFunc A B f
-- hg : IsFunc A B g
rw [lambda_eta hf, lambda_eta hg, lambda_ext_iff]
```

This ends up in lifting a proof of extensionality for ZFC functions to standard extensionality of Lean functions. Full details of the proof are available in the artifacts.

3.3 Automation of side-conditions

Manipulating relations in a set-theoretic style generates repetitive proof obligations regarding relations and partial/total functions. As already mentioned, the library provides three lightweight tactics whose purpose is to try to solve such obligations *structurally*:

- `zrel`: relational goals $R \subseteq A \times B$;
- `zpfun`: partial functionality goals $\mathsf{IsPFunc}(f, A, B)$;
- `zfun`: total functionality goals $\mathsf{IsFunc}(A, B, f)$.

All three tactics work similarly: they first search among local hypotheses for witnesses of the required properties, or apply standard theorems (e.g., composition of functions is a function) to reduce the goal to simpler subgoals.

They may then recursively call each other and backtrack as needed when application of a theorem fails. In order to reduce search time and keep automation usable at runtime, some theorems are selected and tagged with relevant attributes—called `zrel`, `zpfun`, and `zfun` for consistency with the tactic names—so that they can be found by the corresponding tactic. This design is illustrated in Figure 1. This also allows all three tactics to be extended with new rules as needed. When an automatic discharge fails, it falls back to the user for manual proof. One can also pass explicit proofs to definitions and theorems instead of relying on automation, even when the tactics would succeed. We show a few examples of typical such theorems (with proofs omitted) in Listing 2.

Given this infrastructure, we then leverage Lean's *automatic parameters* to pass default tactics in partial definitions depending on side-conditions, so that those side-conditions are automatically discharged when possible. Automatic parameters are very similar to default

Listing 2 Examples of theorems tagged for automation.

```
@[zrel] -- converse of a relation is a relation
theorem subset_prod_inv {R A B : ZFSet} (hR : R ⊆ A × B) :
  R⁻¹ ⊆ B × A := ⋯
@[zpfun] -- identity is a partial function
theorem Id.IsPFunc {A : ZFSet} : (𝟙A).IsPFunc A A := ⋯
@[zfun] -- composition of total functions is a total function
theorem IsFunc_of_composition_IsFunc {g f : ZFSet} {A B C : ZFSet}
  (hg : B.IsFunc C g) (hf : A.IsFunc B f) :
    A.IsFunc C (composition g f A B C) := ⋯
```

Listing 3 Examples of simplification rules for relations and functions.

```
@[simp] theorem Image_empty {R A B : ZFSet} (hR : R ⊆ A.prod B) : R[∅] = ∅ := ⋯
@[simp] theorem range_Id {A : ZFSet} : (𝟙A).Range = A := ⋯
@[simp] theorem Image_of_composition_inv_self_of_bijective {f A B X : ZFSet}
  {hf : A.IsFunc B f} (hf : f.IsBijective) (hX : X ⊆ A) : f⁻¹[f[X]] = X := ⋯
```

parameters, but are specific to tactics: they try to discharge the goal using the provided default tactic if the argument is not supplied explicitly, and fail silently otherwise, leaving the user to provide a manual proof. Furthermore, those side conditions often depend on parameters that are kept implicit (e.g., the sets A and B in $R \subseteq A \times B$); they are also inferred automatically by Lean. This ultimately enables working with these definitions—most of the time—without worrying about side conditions or implicit parameters, which is the intended purpose of the ZFLean framework.

Listing 2 illustrates this already: the converse relation R^{-1} in theorem `subset_prod_inv` is actually a relation between B and A, which are implicit at this level, and relies on the side-condition $R \subseteq A \times B$, automatically discharged by `zrel`. This principle will be further exemplified in Section 5. We also tag relevant equality theorems with the `simp` attribute, so that they can be used by Lean's `simp` tactic for rewriting. A few examples are shown in Listing 3, with proofs omitted.

Overall, this calculus provides a compact, rewriting-friendly and easily extensible interface for extensional reasoning with relations and functions *inside* ZFC, while hiding routine side-conditions behind small, domain-specific tactics.

4 Canonical Constructions and Universal Properties

The relational calculus of Section 3 already provides a stable framework for manipulating relations and (partial) functions internally in the ZFC model. We now build a library of canonical ZFC objects and expose their characteristic properties in a form designed to be usable in practice, compositional, and integrated with the relational calculus.

4.1 Canonical objects

Boolean algebra

We define a canonical two-element set-theoretic object $\mathbb{B} : \mathsf{ZFSet} \triangleq \{\bot, \top\}$ and its usual associated algebraic structure, with accompanying lemmas and notations. Definitions are

Listing 4 Case analysis on internal Booleans.

```
@[cases_eliminator]
def ZFBool.casesOn {motive : ZFBool → Sort _} (p : ZFBool)
  (false : motive ⊥) (true : motive ⊤) : motive p
```

Listing 5 Some simplification lemmas for ZFC Booleans.

```
theorem and_comm  (p q : ZFBool)   : p ⋀ q = q ⋀ p := ⋯
theorem and_assoc (p q r : ZFBool) : p ⋀ q ⋀ r = p ⋀ (q ⋀ r) := ⋯
@[simp]
theorem and_true  (p : ZFBool)     : p ⋀ ⊤ = p := ⋯
theorem and_intro (p q : ZFBool)   : p = ⊤ ∧ q = ⊤ → p ⋀ q = ⊤ := ⋯
```

standard: $\bot \triangleq \varnothing$, $\top \triangleq \{\varnothing\}$, conjunction as intersection, disjunction as union, and negation as set-theoretic complement relative to $\mathbb{B}$.

▶ Note 4.1 (Subtypes as extensional types). For any set $S : \mathsf{ZFSet}$, its extension can be exposed as a Lean subtype $\{x : \mathsf{ZFSet} \mid x \in S\}$. Concretely, an element $e : \{x : \mathsf{ZFSet} \mid x \in S\}$ can be coerced back to a set, and the stored membership proof is still accessible. Conversely, a raw $e : \mathsf{ZFSet}$ equipped with $h : e \in S$ can be packaged as $\langle e, h\rangle : \{x : \mathsf{ZFSet} \mid x \in S\}$, and equality of subtype values reduces to equality of underlying sets via extensionality and the proofs are irrelevant.

We define a proper type $\mathsf{ZFBool} \triangleq \{x : \mathsf{ZFSet} \mid x \in \mathbb{B}\}$ leveraging Lean's *subtypes*, which is *not* a set and must be read as "the type of sets belonging to $\mathbb{B}$". Our algebraic structure is then elaborated upon this type. First, we define a *case eliminator* casesOn for ZFBool, enabling branching upon Booleans—that is, case analysis via the `cases` tactic, or case splitting in definitions. Its type is given in Listing 4.

Then, we define the usual boolean operations (conjunction, disjunction, negation, implication, etc.) as operations on ZFBool, ensuring that the result of each operation is again a member of $\mathbb{B}$ by construction.

▶ **Definition 4.2** (Conjunction). *Given $\langle p, h_p\rangle, \langle q, h_q\rangle : \mathsf{ZFBool}$, their conjunction $p \bigwedge q : \mathsf{ZFBool}$ is defined as the pair $\langle p \cap q, h\rangle$ where $h : p \cap q \in \mathbb{B}$ is a proof obtained by case analysis on p and q; this yields four cases, all easily discharged.*

Similar definitions are provided for disjunction ($\bigvee$) and negation (not), together with the expected algebraic laws such as commutativity, associativity, distributivity, neutral elements and simplification lemmas. Listing 5 shows some of those laws.

Natural numbers

Mathlib already provides a set-theoretic object ω (the first infinite von Neumann ordinal) inside the ZFC model. Reusing it is technically harmless: nothing in the development relies on a peculiarity of our presentation, and one can obtain a natural-number structure from ω by working with its elements as (finite) von Neumann ordinals. We nonetheless re-derive $\mathbb{N}$ directly from the axiom of infinity as the least inductive set—thereby keeping the development self-contained—because ω is introduced a priori as an *ordinal* object geared towards ordinal reasoning rather than towards a minimal natural-numbers interface. Our

Listing 6 Recursor for ZFNat.

```
@[induction_eliminator]
def rec.{u} {motive : ZFNat → Sort u} (n : ZFNat)
  (zero : motive 0) (succ : Π x, motive x → motive (succ x)) : motive n := ⋯
```

Listing 7 Basic arithmetic operations on ZFNat and some associated facts.

```
def pred  (m : ZFNat) : ZFNat := ZFNat.rec m 0 (fun x _ ↦ x)
def add (n m : ZFNat) : ZFNat := ZFNat.rec n m (fun _ ↦ succ)
def sub (n m : ZFNat) : ZFNat := ZFNat.rec m n (fun _ ↦ pred)
def mul (n m : ZFNat) : ZFNat := ZFNat.rec n 0 (fun _ ↦ (· + m))
--
theorem sub_add_distrib {n m k : ZFNat} : n - (m + k) = n - m - k := ⋯
theorem left_distrib    {n m k : ZFNat} : n * (m + k) = n * m + n * k := ⋯
theorem mul_lt_mono     {n m k : ZFNat} (h : 0 < k) : n < m → k*n < k*m := ⋯
```

construction explicitly targets natural numbers: it fixes the successor, recursion/induction principle, and arithmetic operations with definitional equalities.

We still achieve a von Neumann-like representation of (transitive) natural numbers, where each natural number is represented as the set of all its predecessors.

▶ **Definition 4.3** (Inductive set). *We call a set* $X : \mathsf{ZFSet}$ inductive *when*

1. $\varnothing \in X$;
2. X *is closed under the operation* $n \mapsto n \cup \{n\}$, *i.e.* $\forall\, n \in X,\ n \cup \{n\} \in X$.

The axiom of infinity guarantees the existence of at least one inductive (infinite) set, which we classically choose and denote by S^∞. We then define the set of natural numbers as follows.

▶ **Definition 4.4.** $\mathbb{N}$ *is defined as the smallest inductive set:*

$$\mathbb{N} \triangleq \bigcap \{X \subseteq S^\infty \mid X \text{ is inductive}\}$$

▶ Remark 4.5. Definition 4.4 is actually agnostic to the choice of S^∞.

In ZFLean, we expose naturals as the subtype $\mathsf{ZFNat} \triangleq \{n : \mathsf{ZFSet} \,//\, n \in \mathbb{N}\}$, with $0_{\mathbb{N}} := \varnothing$ and $1_{\mathbb{N}} \triangleq \mathsf{succ}(0_{\mathbb{N}})$ where the successor function is defined as:

$$\begin{array}{rccc} \mathsf{succ}: & \mathsf{ZFNat} & \to & \mathsf{ZFNat} \\ & \langle n, h\rangle & \mapsto & \langle n \cup \{n\}, h'\rangle \end{array} \quad \text{where } h' : n \cup \{n\} \in \mathbb{N} \text{ is derived from } h : n \in \mathbb{N}.$$

This definition yields the expected inductive structure on $\mathbb{N}$, with each natural number n being the set of all smaller natural numbers. This offers a straightforward definition of the standard (strict) total ordering on ZFNat as set membership: $\langle m, h_m\rangle < \langle n, h_n\rangle \triangleq m \in n$, which we instantiate in Lean and derive the expected properties (irreflexivity, transitivity, trichotomy, etc.). We also define the non-strict order as usual: $m \le n \triangleq m < n \vee m = n$.

We then derive a well-founded recursion principle—as well as weak/strong induction—as a fixpoint, using that succ induces a well-founded relation on ZFNat. The type of ZFNat.rec is shown in Listing 6 and contains the two usual branches for zero and successor. We indicate to Lean that this is an *induction eliminator* so that the `induction` tactic can leverage it. Finally, we define standard arithmetic operations by primitive recursion, together with convenient

Listing 8 Example of ring equality on ZFNat.

```
example (a b : ZFNat) : (a + b)^2 = a^2 + 2*a*b + b^2 := by ring
```

notations and a furnished library of arithmetic lemmas. A few examples of such theorems are shown in Listing 7. We eventually equip ZFNat with a *commutative semiring* structure and allow to leverage Mathlib's `ring` tactic, which can automatically prove equalities in semirings by normalization. Listing 8 shows that `ring` can be used seamlessly on ZFNat and is able to prove the binomial squares identity automatically.

Integers and rationals

We follow standard constructions to build integers as equivalence classes of pairs of naturals under the relation $(a, b) \sim_{\mathbb{Z}} (c, d) \iff a + d = b + c$ for all $a, b, c, d : \mathsf{ZFNat}$. Contrary to naturals however, we do not primarily expose integers as a set, but as quotient type.

▶ **Definition 4.6** (Ring of integers). *We define the set of integers as the quotient:*

$$\mathsf{ZFInt} \triangleq (\mathsf{ZFNat} \times \mathsf{ZFNat})/\sim_{\mathbb{Z}}$$

We then define $0_{\mathbb{Z}} \triangleq [(0_{\mathbb{N}}, 0_{\mathbb{N}})]_{\sim_{\mathbb{Z}}}$, $1_{\mathbb{Z}} \triangleq [(1_{\mathbb{N}}, 0_{\mathbb{N}})]_{\sim_{\mathbb{Z}}}$ *(equivalence classes of the representatives), and the usual addition, subtraction and multiplication operations on* ZFInt *by lifting the corresponding operations on representatives and instantiate a* commutative ring *structure* $\langle \mathsf{ZFInt}, +, * \rangle$ *with the usual ordering.*

We then define a set $\mathbb{Z}$ *of canonical representatives of integers as the union as follows*

$$\mathbb{Z} \triangleq \mathbb{N} \times \{0_{\mathbb{N}}\} \cup \{0_{\mathbb{N}}\} \times \mathbb{N}$$

and prove that the subtype built from the extension of the set $\mathbb{Z}$ *is isomorphic to* ZFInt *(which also provides coercions).*

▶ Remark 4.7. This definition does not imply that $\mathbb{N} \subseteq \mathbb{Z}$ but rather that $\mathbb{N}$ is isomorphic to the set of nonnegative integers. Instead, it views integers as algebraic "distances" between two naturals, so any integer can be represented by infinitely many pairs of naturals. This is particularly useful to define the opposite operation since it simply consists in flipping pairs; subtraction is then directly obtained as addition of the opposite.

Implementing rationals follows a similar pattern, as equivalence classes of pairs of integers under the relation $(a, b) \sim_{\mathbb{Q}} (c, d) \iff a * d = b * c$ for all $a, b, c, d : \mathsf{ZFInt}$ with $b, d \neq 0_{\mathbb{Z}}$. This yields a quotient type $\mathsf{ZFRat} \triangleq (\mathsf{ZFInt} \times \mathsf{ZFInt}^{\star})/\sim_{\mathbb{Q}}$.[4] As with integers, we implement the basic operations: addition, subtraction, and multiplication are lifted from ZFInt, while division is defined via $\sim_{\mathbb{Q}}$. We prove the required properties and ultimately instantiate a *commutative field* structure on $\langle \mathsf{ZFRat}, +, * \rangle$.

Coproducts and options

We also provide canonical set-theoretic representations for coproducts (disjoint unions) and options. These constructions carry less mathematical weight than the previous ones, but can be useful to model sum types and alternative partiality in a set-theoretic context.

[4] $\mathsf{ZFInt}^{\star}$ denotes the type ZFInt without $0_{\mathbb{Z}}$.

▶ **Definition 4.8.** *Given two sets* $A, B : \mathsf{ZFSet}$*, the* sum*, or* coproduct *type* $A \uplus B$ *is defined as the following extension (Lean subtype):*

$$A \uplus B \triangleq \{x : \mathsf{ZFSet} \,//\, x \in (\{\bot\} \times A) \cup (\{\top\} \times B)\}$$

and is equipped with canonical injections

$$\mathsf{inl} : A \to A \uplus B \qquad \textit{and} \qquad \mathsf{inr} : B \to A \uplus B$$

From this, we provide a set-theoretic definition of the *option* type via the coproduct $\mathsf{Option}\, A \triangleq \{\varnothing\} \uplus A$ equipped with expected canonical injections $\mathsf{none} : \mathsf{Option}\, A \triangleq \mathsf{inl}(\varnothing)$ and $\mathsf{some} : A \to \mathsf{Option}\, A \triangleq \mathsf{inr}$ as well as a case eliminator.

4.2 Embeddings, isomorphisms, and transport

Embeddings and isomorphisms

We define embeddings and isomorphisms between sets internally in ZFC.

▶ **Definition 4.9** (Set-level embeddings and isomorphisms)**.** *A set* A *is* embeddable *into a set* B*, denoted* $A \hookrightarrow^{\mathsf{z}} B$*, when there exists an injective total function from* A *to* B*:*

$$A \hookrightarrow^{\mathsf{z}} B \triangleq \exists\, (f : \mathsf{ZFSet})\, (h_f : \mathsf{IsFunc}(A, B, f)),\ \mathsf{Injective}(f)$$

A and B are said to be isomorphic*, denoted* $A \cong^{\mathsf{z}} B$*, when there exists a bijective function*[5] *from A to B:*

$$A \cong^{\mathsf{z}} B \triangleq \exists\, (f : \mathsf{ZFSet})\, (h_f : \mathsf{IsFunc}(A, B, f)),\ \mathsf{Bijective}(f)$$

▶ Remark 4.10. $\mathsf{Injective}(f)$ and $\mathsf{Bijective}(f)$ expect f to be a total function from A to B, which is given by h_f; this is handled by our automation layer—here, by the `zfun` tactic.

The relation $\cong^{\mathsf{z}}$ is shown to be an equivalence relation on sets, and $\hookrightarrow^{\mathsf{z}}$ is shown to be antisymmetric up to isomorphism, a property known as the *Cantor-Schröder-Bernstein theorem* [4].

▶ **Theorem 4.11** (Cantor-Schröder-Bernstein)**.** *For any sets A, B, if $A \hookrightarrow^{\mathsf{z}} B$ and $B \hookrightarrow^{\mathsf{z}} A$, then $A \cong^{\mathsf{z}} B$, i.e. in ZFLean:*

```
theorem isIso_of_biembedding {A B : ZFSet} (h : A ↪ᶻ B) (h' : B ↪ᶻ A) :
  A ≅ᶻ B := ⋯
```

Proof. The proof is available in the artifacts and follows a classical pattern using successive sets of *stable* elements under the two embeddings. ◀

Bridges to Lean types and transport along isomorphisms

To interoperate with Lean/Mathlib infrastructure, our library provides explicit "bridges" between canonical ZFC objects and native types, notably we prove that our constructions are isomorphic—here, at type level, denoted $\simeq$—to their Lean counterparts, namely

$$\mathsf{ZFBool} \simeq \mathsf{Bool}, \qquad \mathsf{ZFNat} \simeq \mathsf{Nat}, \qquad \mathsf{ZFInt} \simeq \mathsf{Int},$$
$$A \uplus B \simeq \{x : \mathsf{ZFSet} \,//\, x \in A\} \oplus \{y : \mathsf{ZFSet} \,//\, y \in B\} \quad \text{(Lean's sum type)}$$
$$\mathsf{ZFSet.Option}\, A \simeq \mathsf{Option}\, \{x : \mathsf{ZFSet} \,//\, x \in A\}$$

[5] In the category of sets, isomorphisms are the bijections.

We also instantiate coercions when relevant, so that users can seamlessly switch between ZFC and Lean/Mathlib representations, as illustrated in the next example.

▶ **Example 4.12.** Proving that conjunction distributes over disjunction on ZFBool, i.e. that for any $p, q, r : \mathsf{ZFBool}$, $p \wedge^{\mathsf{z}} (q \vee^{\mathsf{z}} r) = (p \wedge^{\mathsf{z}} q) \vee^{\mathsf{z}} (p \wedge^{\mathsf{z}} r)$ holds, can be done by transporting the corresponding theorem from Lean's native boolean type Bool; see theorem `and_or_distrib_left` in the artifacts for full proof details.

5 Case Study and Evaluation

Internally, all constructions developed so far in the ZF model are *sets*, including functions, relations, naturals, etc. This section illustrates the intended *user workflow* in ZFLean on a classical result that exercises most of the infrastructure developed above, with an attempt to demonstrate how the interface is designed so that these implementation details rarely surface. The chosen example is the *currying isomorphism*, which is representative because it combines nested abstraction, function application and transformations, while remaining mathematically standard and proof-script friendly.

5.1 Currying isomorphism

▶ **Theorem 5.1** (Currying isomorphism). *For any sets A, B, C, the following holds:*

$$C^{A\times B} \cong^{\mathsf{z}} (C^B)^A$$

We use two mutually inverse functions to build the isomorphism, corresponding to the usual (un)currying operation. Those functions are defined as follows in ZFLean.

▶ **Definition 5.2** (Currying and uncurrying). *We define set-level functions* $\mathsf{curry}^{\mathsf{z}} : \mathsf{ZFSet}$ *and* $\mathsf{uncurry}^{\mathsf{z}} : \mathsf{ZFSet}$ *as follows in ZFLean, mirroring the mathematical definition and syntax:*

$$\mathsf{curry}^{\mathsf{z}} \triangleq \begin{pmatrix} \lambda^{\mathsf{z}} & : & C^{A\times B} & \to & (C^B)^A \\ & | & f & \mapsto & \mathcal{C}(f) \end{pmatrix}, \quad \mathsf{uncurry}^{\mathsf{z}} \triangleq \begin{pmatrix} \lambda^{\mathsf{z}} & : & (C^B)^A & \to & C^{A\times B} \\ & | & f & \mapsto & \mathcal{U}(f) \end{pmatrix}$$

where

$$\mathcal{C}(f) \triangleq \begin{pmatrix} \lambda^{\mathsf{z}} & : & A & \to & C^B \\ & | & a & \mapsto & \begin{pmatrix} \lambda^{\mathsf{z}} & : & B & \to & C \\ & | & b & \mapsto & @^{\mathsf{z}} f(\langle (a,b), \cdots \rangle) \end{pmatrix} \end{pmatrix}$$

and

$$\mathcal{U}(f) \triangleq \begin{pmatrix} \lambda^{\mathsf{z}} & : & A \times B & \to & C \\ & | & (a,b) & \mapsto & @^{\mathsf{z}}\,(@^{\mathsf{z}} f(\langle a, \cdots \rangle))\,(\langle b, \cdots \rangle) \end{pmatrix}$$

We prove that $\mathsf{curry}^{\mathsf{z}}$ *and* $\mathsf{uncurry}^{\mathsf{z}}$ *are total functions on their respective domains, so that automation can pick those facts up when needed:* $\mathsf{IsFunc}(C^{A\times B}, (C^B)^A, \mathsf{curry}^{\mathsf{z}})$ *and* $\mathsf{IsFunc}((C^B)^A, C^{A\times B}, \mathsf{uncurry}^{\mathsf{z}})$.

We then prove that these two functions are mutually inverse; this is the core of the proof of Theorem 5.1.

▶ **Lemma 5.3.**

$$\mathsf{curry}^{\mathsf{z}} \circ^{\mathsf{z}} \mathsf{uncurry}^{\mathsf{z}} = \mathbb{1}_{(C^B)^A} \quad \textit{and} \quad \mathsf{uncurry}^{\mathsf{z}} \circ^{\mathsf{z}} \mathsf{curry}^{\mathsf{z}} = \mathbb{1}_{C^{A\times B}}$$

Proof. We sketch the proof of the first equality; the second one follows a similar pattern. Full proof details are available in the artifacts. First, note that both sides contain side-conditions ensuring that everything is well-defined, all automatically discharged by the automation layer. We first reduce the equality of functions to a pointwise equality using function extensionality, using theorem `is_func_ext_iff` from Example 3.5. Then, let $f \in (C^B)^A$ be an arbitrary set-level curried function; we need to show the following equality:

$$@^{\mathsf{z}}(\mathsf{curry}^{\mathsf{z}} \circ^{\mathsf{z}} \mathsf{uncurry}^{\mathsf{z}})(\langle f, \cdots\rangle) = @^{\mathsf{z}}\mathbb{1}_{(C^B)^A}(\langle f, \cdots\rangle)$$

The right-hand side reduces to f by definition of the identity function and using the following theorem:

```
theorem fapply_Id {A x : ZFSet} (hx : x ∈ A) : @ᶻ𝟙A ⟨x, ⋯⟩ = ⟨x, hx⟩ := ⋯
```

We then use commutativity of evaluation and composition in the left-hand side using the following theorem:

```
theorem fapply_composition {g f : ZFSet} {A B C : ZFSet}
  (hg : B.IsFunc C g) (hf : A.IsFunc B f) {x : ZFSet} (xA : x ∈ A) :
  @ᶻ(g ∘ᶻ f) ⟨x, ⋯⟩ = @ᶻg ⟨@ᶻf ⟨x, ⋯⟩, ⋯⟩ := ⋯
```

This reduces the left-hand side to $@^{\mathsf{z}}\mathsf{curry}^{\mathsf{z}}(\langle @^{\mathsf{z}}\mathsf{uncurry}^{\mathsf{z}}(\langle f, \_\rangle), \_\rangle)$. Unfolding the definitions of $\mathsf{curry}^{\mathsf{z}}$ and $\mathsf{uncurry}^{\mathsf{z}}$ and applying β-reduction twice from Theorem 3.4 then yields the following equality to prove:

$$\mathcal{C}(\mathcal{U}(f)) = f$$

Again, we leverage the framework's functional extensionality theorem and apply it twice to reduce this equality to a pointwise one for any $a \in A$ and $b \in B$:

$$@^{\mathsf{z}}(@^{\mathsf{z}}\mathcal{C}(\mathcal{U}(f))(\langle a, \cdots\rangle))(\langle b, \cdots\rangle) = @^{\mathsf{z}}(@^{\mathsf{z}}f(\langle a, \cdots\rangle))(\langle b, \cdots\rangle)$$

Further β-reductions are applied to the unfolded definition of $\mathcal{C}$, yielding:

$$@^{\mathsf{z}}\left(\begin{array}{cccc} \lambda^{\mathsf{z}} & : & B & \to & C \\ & | & b' & \mapsto & @^{\mathsf{z}}\mathcal{U}(f)(\langle(a, b'), \cdots\rangle) \end{array}\right)(\langle b, \cdots\rangle) = @^{\mathsf{z}}(@^{\mathsf{z}}f(\langle a, \cdots\rangle))(\langle b, \cdots\rangle)$$

and to the unfolded definition of $\mathcal{U}$ as well, reducing the goal to the trivial equality $@^{\mathsf{z}}(@^{\mathsf{z}}f(\langle a, \cdots\rangle))(\langle b, \cdots\rangle) = @^{\mathsf{z}}(@^{\mathsf{z}}f(\langle a, \cdots\rangle))(\langle b, \cdots\rangle)$. ◀

The proof of Theorem 5.1 then boils down to applying the following theorem with the equalities from Lemma 5.3:

```
theorem isIso_of_two_sided_inverse {A B : ZFSet} {f g : ZFSet}
  {hf : A.IsFunc B f} {hg : B.IsFunc A g}
  (left_inv : g ∘ᶻ f = 𝟙A) (right_inv : f ∘ᶻ g = 𝟙B) : A ≅ᶻ B := ⋯
```

All routine side-conditions ("is a partial/total function", "is a relation") are discharged structurally and automatically by `zrel`/`zpfun`/`zfun` through automatic parameters and the appropriate attribute-tagged closure lemmas. The skeleton of the proof script shown in Listing 9 closely follows the mathematical argument described above. At the proof-script level, the role of the automation layer is not to perform proof search, but to *erase boilerplate*: the user writes the mathematical argument, and the automation layer discharges the ubiquitous well-formedness obligations generated by set-level definitions and rewriting.

Listing 9 Proof skeleton of the currying isomorphism.

```
theorem isIso_curry {A B C : ZFSet} : (A × B).funs C ≅ᶻ A.funs (B.funs C) := by
  have l_inv : uncurry ∘ᶻ curry = 𝟙((A × B).funs C) := ⋯
  have r_inv : curry ∘ᶻ uncurry = 𝟙(A.funs (B.funs C)) := by
    rw [is_func_ext_iff]
    intro f hf
    unfold curry uncurry
    rw [fapply_Id, fapply_composition, fapply_lambda ⋯ ⋯, fapply_lambda ⋯ ⋯]
    ⋯
  exact isIso_of_two_sided_inverse l_inv r_inv
```

Table 1 Artifact sizes in lines of Lean code (LoC) and number of theorems, grouped by component.

Component (modules)	LoC	#Theorems
Relational calculus	2,534	172
Embeddings, isomorphisms	1,626	31
Other canonicals (Booleans, sums, rationals)	1,362	153
Naturals	1,260	199
Integers	1,225	151
Core glue and automation	317	0
Total	**8,324**	**706**

5.2 Size of the artifact

The artifact is a self-contained Lean 4 library layered on top of Mathlib's ZFC model consisting of 8,324 lines of code (LoC), organized in a few modules summarized in Table 1, and containing 706 proved theorems. The relational calculus is the largest component, representing about one third of the codebase.

6 Related Work

We position our contributions with respect to set-theoretic developments in interactive provers such as Isabelle and Rocq, to Mathlib's ZFC model, and to categorical/set-theoretic bases.

Isabelle/ZF and ZFC in Isabelle/HOL

Isabelle includes a classical first-order logic (FOL) object logic with Zermelo–Fraenkel set theory as an object-level theory, commonly referred to as Isabelle/ZF [8]. In this approach, set-theoretic reasoning happens inside the ZF object logic, with membership and extensionality primitives and large portions of mathematics developed directly at the set level. While powerful, ZF lives as a separate object logic from Isabelle's higher-order logic (HOL). This separation historically delivered deep set-theoretic developments (ordinals, constructibility, etc.), but makes it less convenient to reuse HOL automation, type classes, and libraries in mixed developments.

A complementary line encodes ZFC *within* Isabelle/HOL by introducing a HOL type of sets together with an elementhood relation and basic operations, with an emphasis on close integration with HOL's automation and type classes [9]. In this setting, sets are first-class HOL values; proofs benefit from standard HOL tooling and can interoperate smoothly with typed libraries. Conceptually, our Lean framework plays a similar bridging role: we keep ZFC-level objects and proofs, but we provide explicit bridges to Lean's native

types so that tactics, typeclasses, and algebraic infrastructure can be reused where they pay off. Our distinct contribution is a *rewriting-friendly relational calculus* tailored to set-level developments, whereas the HOL encodings typically emphasize a minimal new notation surface and type-class integration.

Rocq (ZFC, IZF)

Rocq has multiple set-theoretic developments with different aims. One family (e.g. `coq-zfc`) encodes ZFC à la Aczel with primitive membership and extensional equality, and then *proves* the ZFC axioms as theorems of the Rocq calculus [1]. Another strand (e.g. `coq-izf`, constructive or intuitionistic set theories) targets constructive metatheory or models of type theory, sometimes via pointed-graph semantics [7] or Tarski–Grothendieck universes [3]. These developments showcase that strong set-level mathematics is possible inside a dependent type theory, but integration with Rocq's typed algebraic hierarchy and automation usually remains more ad-hoc.

ETCS and categorical foundations

Lawvere's Elementary Theory of the Category of Sets (ETCS) axiomatizes the (well-pointed) category of sets with finite limits, cartesian closure, a natural numbers object, and (typically) choice; equality is extensional at the *morphism* level and "up to isomorphism" reasoning is built in from the start [5]. ETCS is weaker than ZFC and is rather a metatheoretic object than a framework for set-level development. In principle though, ETCS could be pursued in Lean via category-theoretic libraries.

Overall, our Lean framework shares with the above works the goal of enabling set-level mathematics inside a typed proof assistant, namely Lean, with smooth interoperability with typed libraries and automation.

7 Conclusion

We advocated a practical way to develop mathematics in ZFC inside Lean 4 by engineering a relational calculus, establishing universal properties with transport along isomorphisms, and building bridges to native libraries. The resulting layer reduces friction when reasoning extensionally and remains compatible with typed developments. Future work includes extending the library with more canonical set-theoretic constructions, such as building reals via Cauchy sequences or Dedekind cuts (or both), and improving automation to further reduce boilerplate in proofs. One might push the framework further and develop more mathematics inside it, for instance basic analysis and related results.

Declaration of AI Use

We declare that no generative AI tools or large language models (LLMs) were used in the development of the tool, the Lean implementation, or the writing of this paper.

References

1 Peter Aczel. The type theoretic interpretation of constructive set theory. In Angus Macintyre, Leszek Pacholski, and Jeff Paris, editors, *Logic Colloquium '77*, volume 96 of *Studies in Logic and the Foundations of Mathematics*, pages 55–66. Elsevier, 1978. doi:10.1016/S0049-237X(08)71989-X.

2 Peter Aczel and Michael Rathjen. Notes on constructive set theory. Technical report, Institut Mittag-Leffler (The Royal Swedish Academy of Sciences), 2001.

3 Bruno Barras. Sets in coq, coq in sets. *Journal of Formalized Reasoning*, 3(1):29–48, 2010. URL: https://doi.org/10.6092/issn.1972-5787/1695, doi:10.6092/ISSN.1972-5787/1695.

4 Georg Cantor. Grundlagen einer allgemeinen Mannigfaltigkeitslehre. *Mathematische Annalen*, 46:481–512, 1895.

5 F. William Lawvere. An elementary theory of the category of sets. *Proceedings of the National Academy of Sciences of the USA*, 52(6):1506–1511, 1964. URL: https://www.pnas.org/doi/pdf/10.1073/pnas.52.6.1506, doi:10.1073/pnas.52.6.1506.

6 Mathlib Community. Mathlib documentation: ZFC model of sets (`PSet`, `ZFSet`), 2026. URL: https://leanprover-community.github.io/mathlib4_docs/Mathlib/SetTheory/ZFC/Basic.html.

7 Alexandre Miquel. lamda-z: Zermelo's set theory as a PTS with 4 sorts. In Jean-Christophe Filliâtre, Christine Paulin-Mohring, and Benjamin Werner, editors, *Types for Proofs and Programs, International Workshop, TYPES 2004, Jouy-en-Josas, France, December 15-18, 2004, Revised Selected Papers*, volume 3839 of *Lecture Notes in Computer Science*, pages 232–251. Springer, 2004. doi:10.1007/11617990_15.

8 Lawrence C. Paulson. Set theory for verification. I: from foundations to functions. *Journal of Automated Reasoning*, 11(3):353–389, December 1993. doi:10.1007/BF00881873.

9 Lawrence C. Paulson. Zermelo Fraenkel set theory in higher-order logic. *Archive of Formal Proofs*, October 2019. https://isa-afp.org/entries/ZFC_in_HOL.html, Formal proof development.

10 Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. Dependent types and multi-monadic effects in F*. *SIGPLAN Not.*, 51(1):256–270, January 2016. doi:10.1145/2914770.2837655.

11 The Mathlib Community. The lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '20)*, 2020. URL: https://leanprover-community.github.io/papers/mathlib-paper.pdf, doi:10.1145/3372885.3373824.