A Lean proof of de Zolt's theorem in higher dimension

Bruno Cuconato* Edward Hermann Haeusler

Departamento de Informática, PUC-Rio
{bclaro,hermann}@inf.puc-rio.br

We present a proof of de Zolt's theorem in tridimensional space, formalized with the assistance of the Lean prover [11, 12]. While de Zolt's theorem is not provable in ZFC, it is provable in Z_p, the system proposed by Giovannini *et al.* in [9, §5]. The Z_p system is a continuation of the work by Giovannini *et al.* in [8], where an abstract algebraic proof of de Zolt's theorem for bidimensional space is presented.

This document is structured as follows: we start by describing the Lean proof assistant, then we show how we define the geometric objects we will be working with in it; we then present the Z_p system as represented in Lean; finally, we show the Lean proof of de Zolt's theorem.

1 The Lean prover

Lean is a pure functional programming language designed to aid formal proofs. It is based in type theory like other similar endeavours like Coq [14] and Agda [13], more specifically the Calculus of Inductive Constructions (CIC) of Coquand and Huet [4].

*I thank CNPq for the financial support (process number 157833/2019–4).

What differentiates Lean from most other theorem provers is its sprawling mathematics library, mathlib [3]. It contains more than a hundred thousand theorems and about half as many definitions, totalling more than a million lines of code contributed by 300 people. Besides the community-building efforts of mathematicians like Jeremy Avigad, Patrick Massot and Kevin Buzzard, the reason for this success is believed to be Lean's extensibility, which has been increased even more in its fourth version [12] with the addition of hygienic macros [16] inspired by the Racket programming language.

Not only has Lean been at the frontier of machine-checked mathematical proofs (for a sample of recent developments, see [10, 7, 5, 1, 2, 6]), but it has also been a platform for innovations in the field of programming languages, specially functional ones. Lean 4 introduced an improvement over the traditional *do* notation that is used as syntactic sugar for imperative-style programming [15], allowing such features as local mutation, early return, and iteration. None of these features are currently supported by the Haskell language, which introduced the idea. Another contribution to language design was a new technique for reference counting in purely functional programming languages [17]. Any reference-couting implementation makes a garbage collector unnecessary, and this one improves on them by reducing the number of reference updates and providing a new memory-reclaiming algorithm for nonshared values.

2 Geometric objects in Lean

We take geometric points as given, so we define them as an opaque Lean type (henceforth written as `Point`). This means they are not explicitly described, and we can't 'look inside' them.

```
structure Point : Type
```

The treatment given to `Point` is exceptional: all the other geometric objects are represented in Lean by their constructions (or deconstructions, depending on how you look at it).

```
structure Segment : Type where
  p1 : Point
  p2 : Point
  neq : p1 ≠ p2
```

A pair of points form a `Segment`; in Lean, this is a structure containing the two points, together with a proof that they are different from each other (i.e., not the same point).

The type of `PolySegment` is isomorphic to a list of `Segment`, having a constructor for the empty `PolySegment`, a constructor that raises a `Segment` into a singleton `PolySegment`, and one that joins two values of `PolySegment` into one.

```
inductive PolySegment : Type where
| ₀s : PolySegment
| ₁s : Segment → PolySegment
| ₂s : PolySegment → PolySegment → PolySegment

opaque IsJordan : PolySegment → PolySegment → Prop
```

A `Face` is a pair of `PolySegment` and a proof that they form a Jordan curve. For this purpose, a Jordan curve is defined opaquely as a predicate `IsJordan` over a pair of `PolySegment`. A `PolyFace` is isomorphic to a list of `Face`, having a constructor for the empty `PolyFace`, a constructor that raises a `Face` into a singleton `PolyFace`, and one that joins two values of type `PolyFace` into one.

```
structure Face : Type where
  s1 : PolySegment
  s2 : PolySegment
  jordan : IsJordan s1 s2

inductive PolyFace : Type where
| ₀f : PolyFace
```

```
| ₁f : Face → PolyFace
| ₂f : PolyFace → PolyFace → PolyFace

opaque IsClosed : PolyFace → PolyFace → Prop
```

A `Volume` is a pair of `PolyFace` together with a proof that they form a closed volume. For this, a closed volume is defined opaquely as a predicate `IsClosed` over a pair of `PolyFace`. A `PolyVolume` is isomorphic to a list of `Volume`, having a constructor for the empty `PolyVolume`, a constructor that raises a `Volume` into a singleton `PolyVolume`, and one that joins two values of type `PolyVolume` into one.

```
structure Volume : Type where
  vol1 : PolyFace
  vol2 : PolyFace
  closed : IsClosed vol1 vol2

inductive PolyVolume where
| ₀v : PolyVolume
| ₁v : Volume → PolyVolume
| ₂v : PolyVolume → PolyVolume → PolyVolume
```

3 The Z_p system in Lean

We can classify the rules of the Z_p system in two groups: the ones related to the construction of geometrical objects, and the ones related to comparing them for size.

The first rules are implemented in Lean as a typeclass, of which the types described in 2 are instances.

```
class Zₚ (a : Type u) where
  ε : a
  cmp : a → a → Prop
  join : (p : a) → (q : a) → cmp p q → a
  NonEmpty : a → Prop
```

```
axiom Z_p.empty_left_join {t} [Z_p t] {p : t} ε{p :
   cmp ε p} : join ε p εp = p
axiom Z_p.empty_right_join {t} [Z_p t] {p : t} {εp :
   cmp p ε} : join p ε εp = p
```

In Lean, typeclasses are used as a way to overload notation. Using this case as an example, we may use ε to denote the empty geometric object of any type which is an instance of Z_p, as we do in the deductive rules for Z_p.

The attentive reader will have noticed that in 2 we defined constructors such as `PolyVolume.v2` that may build invalid (not well-formed) geometric objects. For a `PolyVolume` to be well-formed when constructed from a pair of `PolyVolume`, we need them to have an intersection forming a `Face`. To enforce such well-formedness restrictions, the Z_p typeclass defines the `cmp` predicate, that is used as an argument to the `join` function.

Further on, we we will also need a predicate establishing that the geometric object is not the empty one, so we add it here. We also state that joining a Z_p object with the empty object results in the original object, a fact we will employ latter in the proof.

To illustrate how this works in practice, below is the instantiation of `PolyVolume` as part of the Z_p typeclass:

```
opaque PolyVolume.HasFaceIntersection
  : PolyVolume → PolyVolume → Prop

axiom PolyVolume.EmptyAlwaysHasFaceIntersection {v
   : PolyVolume} :
  HasFaceIntersection ₀v v

axiom PolyVolume.HasFaceIntersection_comm {v u :
   PolyVolume} :
  HasFaceIntersection v u → HasFaceIntersection u v
```

```
def PolyVolume.NonEmpty : PolyVolume → Prop
| ₀v => False
| ₁v _ => True
| ₂v _ _ => True

instance : Zₚ PolyVolume where
  ε := PolyVolume.₀v
  cmp := PolyVolume.HasFaceIntersection
  join := λ p v _cmp => PolyVolume.₂v p v
  NonEmpty := PolyVolume.NonEmpty
```

As one can see, we need the auxiliary definition of the `HasFaceIntersection` predicate over a pair of `PolyVolume` to be used as our `cmp` predicate for `PolyVolume`. We also state that having a face intersection is a commutative property, and that the empty `PolyVolume` always has a face intersection with any other `PolyVolume`. To join two `PolyVolumes` in one, we use the regular v_2, but the (otherwise unused) `_cmp` argument guarantees that the result is well-formed. We also define the `NonEmpty` predicate for `PolyVolume` in the obvious way.

The instantiations for the other geometrical objects can be made similarly (with the appropriate predicates), but they are not necessary for the proof of the de Zolt theorem in Z_p.

Now we need to define the rules for comparing Z_p values. $<$ and $\leq$ are defined inductively over pairs of Z_p values:

```
mutual
  variable {t} [Zₚ t]

  inductive Zₚ.le : t → t → Prop where
  | refl {p : t} : le p p
  | ₁le : ∀ {₁p ₁q ₂p ₂q : t}, le ₁p ₁q → le ₂p ₂q
        → (pc : cmp ₁p ₂p) → (qc : cmp ₁q ₂q)
        → le (join ₁p ₂p pc) (join ₁q ₂q qc)
```

```
  inductive Z_p.lt : t → t → Prop
  | ε₁ : ∀ {p q : t}, (pqc : cmp p q) → lt p (join
   p q pqc)
  | ε₂ : ∀ {p q : t}, (pqc : cmp p q) → lt q (join
   p q pqc)
  | ₁lt : ∀ {₁p ₁q ₂p ₂q : t}, lt ₁p ₁q → le ₂p ₂q
        → (pc : cmp ₁p ₂p) → (qc : cmp ₁q ₂q)
        → lt (join ₁p ₂p pc) (join ₁q ₂q qc)
  | ₂lt : ∀ {₁p ₁q ₂p ₂q : t}, le ₁p ₁q → lt ₂p ₂q
        → (pc : cmp ₁p ₂p) → (qc : cmp ₁q ₂q)
        → lt (join ₁p ₂p pc) (join ₁q ₂q qc)
end
```

The constructors of the `le` (for $\leq$) and `lt` (for $<$) types correspond to the Z_p deductive rules of the same name.

4 The formal proof

The final piece we need to prove the de Zolt theorem is the definition of a truncation of a geometric object (in Lean this means any value of a type which is an instance of the Z_p typeclass). We define truncation inductively in the following Lean code:

```
section Truncation
variable {t} [Z_p t]

inductive Z_p.TruncationOf : t → t → Prop where
| ₀t {p : t} : NonEmpty p → TruncationOf ε p
| ₁t {p q r : t} : (pr : cmp p r) → (qr : cmp q r)
    → TruncationOf p q
    → TruncationOf (join p r pr) (join q r qr)

end Truncation
```

A `TruncationOf` value is thus contructed recursively. The base case is that for any non-empty version of a geometric ob-

ject, the empty object is a `TruncationOf` it. So for the case of a `PolyVolume`, for any non-empty `PolyVolume` the empty `PolyVolume` is a `TruncationOf` it. For the inductive case, given three geometric objects, the first of which is a `TruncationOf` the second one, we have that the join of the first object with the third object is a `TruncationOf` the second one with the same third object. This is only true provided we can perform both of these joins, that is, that their results are well-formed geometric objects; this is guaranteed by the two `cmp` arguments.

We are finally ready for the statement of the de Zolt theorem:

```
theorem PolyVolume.zolt {q p : PolyVolume}
  (isTrunc : Z_p.TruncationOf q p)
  : Z_p.lt q p
```

That is, if p, q are `PolyVolume`, and q is a `TruncationOf` p, we have that $q < p$.

The proof is by induction on the `TruncationOf` construction: in the base case, we have that q is the empty `PolyVolume` ε, and so we use the ε_2 rule to show that $\varepsilon < p$. In the inductive case we have that p and q are actually $p'; r$ (`join p' r`) and $q; r$ (`join q r`), and we have a proof of $q' < p'$. With this proof and the trivial proof of $r \leq r$ we can invoke the lt_1 rule to show that $q'; r < p'; r$ □

Or, in Lean:

```
theorem PolyVolume.zolt {q p : PolyVolume}
  (isTrunc : Z_p.TruncationOf q p)
  : Z_p.lt q p :=
  match isTrunc with
  | Z_p.TruncationOf.₀t _ =>
      have εpcmp : Z_p.cmp p ₀v
        := HasFaceIntersection_comm
    EmptyAlwaysHasFaceIntersection
      (Eq.subst Z_p.empty_right_join <| Z_p.ltε₂.
    εpcmp)
  | Z_p.TruncationOf.₁t (p := w) (q := u) (r := r)
```

```
  wrcmp urcmp qIsTruncOfp =>
   have w_lt_u : Z_p.lt w u := zolt qIsTruncOfp
   have r_le_r : Z_p.le r r := Z_p.le.refl
   Z_p.lt.₁lt w_lt_u r_le_r wrcmp urcmp
```

5 The full proof

For the reader's convenience, we reproduce below the full proof without the explanation text.

```
structure Point : Type

structure Segment : Type where
  p1 : Point
  p2 : Point
  neq : p1 ≠ p2

inductive PolySegment : Type where
| ₀s : PolySegment
| ₁s : Segment → PolySegment
| ₂s : PolySegment → PolySegment → PolySegment

opaque IsJordan : PolySegment → PolySegment → Prop

structure Face : Type where
  s1 : PolySegment
  s2 : PolySegment
  jordan : IsJordan s1 s2

inductive PolyFace : Type where
| ₀f : PolyFace
| ₁f : Face → PolyFace
| ₂f : PolyFace → PolyFace → PolyFace

opaque IsClosed : PolyFace → PolyFace → Prop
```

```
structure Volume : Type where
  vol1 : PolyFace
  vol2 : PolyFace
  closed : IsClosed vol1 vol2

inductive PolyVolume where
| ₀v : PolyVolume
| ₁v : Volume → PolyVolume
| ₂v : PolyVolume → PolyVolume → PolyVolume

opaque PolyVolume.HasFaceIntersection
  : PolyVolume → PolyVolume → Prop

axiom PolyVolume.EmptyAlwaysHasFaceIntersection {v
   : PolyVolume} :
  HasFaceIntersection ₀v v

axiom PolyVolume.HasFaceIntersection_comm {v u :
   PolyVolume} :
  HasFaceIntersection v u → HasFaceIntersection u v

def PolyVolume.NonEmpty : PolyVolume → Prop
| ₀v => False
| ₁v _ => True
| ₂v _ _ => True

class Zₚ (a : Type u) where
  ε : a
  cmp : a → a → Prop
  join : (p : a) → (q : a) → cmp p q → a
  NonEmpty : a → Prop

axiom Zₚ.empty_left_join {t} [Zₚ t] {p : t} ε{p :
   cmp ε p} : join ε p εp = p
```

```
axiom Z_p.empty_right_join {t} [Z_p t] {p : t} {εp :
   cmp p ε} : join p ε εp = p

instance : Z_p PolyVolume where
  ε := PolyVolume.₀v
  cmp := PolyVolume.HasFaceIntersection
  join := λ p v _cmp => PolyVolume.₂v p v
  NonEmpty := PolyVolume.NonEmpty

mutual
  variable {t} [Z_p t]

  inductive Z_p.le : t → t → Prop where
  | refl {p : t} : le p p
  | ₁le : ∀ {₁p ₁q ₂p ₂q : t}, le ₁p ₁q → le ₂p ₂q
        → (pc : cmp ₁p ₂p) → (qc : cmp ₁q ₂q)
        → le (join ₁p ₂p pc) (join ₁q ₂q qc)

  inductive Z_p.lt : t → t → Prop
  | ε₁ : ∀ {p q : t}, (pqc : cmp p q) → lt p (join
   p q pqc)
  | ε₂ : ∀ {p q : t}, (pqc : cmp p q) → lt q (join
   p q pqc)
  | ₁lt : ∀ {₁p ₁q ₂p ₂q : t}, lt ₁p ₁q → le ₂p ₂q
        → (pc : cmp ₁p ₂p) → (qc : cmp ₁q ₂q)
        → lt (join ₁p ₂p pc) (join ₁q ₂q qc)
  | ₂lt : ∀ {₁p ₁q ₂p ₂q : t}, le ₁p ₁q → lt ₂p ₂q
        → (pc : cmp ₁p ₂p) → (qc : cmp ₁q ₂q)
        → lt (join ₁p ₂p pc) (join ₁q ₂q qc)
end

section Truncation

  variable {t} [Z_p t]
```

```
  inductive Z_p.TruncationOf : t → t → Prop where
  | ₀t {p : t} : NonEmpty p → TruncationOf ε p
  | ₁t {p q r : t} : (pr : cmp p r) → (qr : cmp q
   r)
      → TruncationOf p q
      → TruncationOf (join p r pr) (join q r qr)

end Truncation

theorem PolyVolume.zolt {q p : PolyVolume}
  (isTrunc : Z_p.TruncationOf q p)
  : Z_p.lt q p :=
  match isTrunc with
  | Z_p.TruncationOf.₀t _ =>
      have εpcmp : Z_p.cmp p ₀v
        := HasFaceIntersection_comm
   EmptyAlwaysHasFaceIntersection
      (Eq.subst Z_p.empty_right_join <| Z_p.ltε₂.
   εpcmp)
  | Z_p.TruncationOf.₁t (p := w) (q := u) (r := r)
   wrcmp urcmp qIsTruncOfp =>
    have w_lt_u : Z_p.lt w u := zolt qIsTruncOfp
    have r_le_r : Z_p.le r r := Z_p.le.refl
    Z_p.lt.₁lt w_lt_u r_le_r wrcmp urcmp
```

References

[1] Anne Baanen et al. “Formalized Class Group Computations and Integral Points on Mordell Elliptic Curves”. In: *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2023. Boston, MA, USA: Association for Computing Machinery, 2023, pp. 47–62. ISBN: 9798400700262. DOI: 10.1145/3573105.3575682. URL: https://doi.org/10.1145/3573105.3575682.

[2] Joshua Clune. “A Formalized Reduction of Keller’s Conjecture”. In: *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2023. Boston, MA, USA: Association for Computing Machinery, 2023, pp. 90–101. ISBN: 9798400700262. DOI: 10.1145/3573105.3575669. URL: https://doi.org/10.1145/3573105.3575669.

[3] The mathlib Community. “The Lean Mathematical Library”. In: *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2020. New Orleans, LA, USA: Association for Computing Machinery, 2020, pp. 367–381. ISBN: 9781450370974. DOI: 10.1145/3372885.3373824. URL: https://doi.org/10.1145/3372885.3373824.

[4] Thierry Coquand and Gérard Huet. “The Calculus of Constructions”. In: *Information and Computation* 76 (1988), pp. 95–120.

[5] Yaël Dillies and Bhavik Mehta. “Formalising Szemerédi’s Regularity Lemma in Lean”. In: *13th International Conference on Interactive Theorem Proving (ITP 2022)*. Ed. by June Andronick and Leonardo de Moura. Vol. 237. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, 9:1–9:19. ISBN: 978-3-95977-252-5. DOI: 10.4230/LIPIcs.ITP.2022.9. URL: https://drops.dagstuhl.de/opus/volltexte/2022/16718.

[6] Floris van Doorn, Patrick Massot, and Oliver Nash. “Formalising the H-Principle and Sphere Eversion”. In: *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2023. Boston, MA, USA: Association for Computing Machinery, 2023, pp. 121–

134. ISBN: 9798400700262. DOI: 10.1145/3573105.3575688. URL: https://doi.org/10.1145/3573105.3575688.

[7] María Inés de Frutos-Fernández. "Formalizing the Ring of Adèles of a Global Field". In: *13th International Conference on Interactive Theorem Proving (ITP 2022)*. Ed. by June Andronick and Leonardo de Moura. Vol. 237. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, 14:1–14:18. ISBN: 978-3-95977-252-5. DOI: 10.4230/LIPIcs.ITP.2022.14. URL: https://drops.dagstuhl.de/opus/volltexte/2022/16723.

[8] Eduardo N. Giovannini et al. "De ZOLT'S POSTULATE: AN ABSTRACT APPROACH". In: *The Review of Symbolic Logic* 15.1 (2022), pp. 197–224. DOI: 10.1017/S1755020319000339.

[9] Eduardo Nicolás Giovannini, Abel Lassalle Casanave, and Edward H Haeusler. "Sobre el Postulado de De Zolt en tres dimensiones". In: *O que nos faz pensar* 29.49 (2021).

[10] Bhavik Mehta. "Formalising Sharkovsky's Theorem (Proof Pearl)". In: *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2023. Boston, MA, USA: Association for Computing Machinery, 2023, pp. 267–274. ISBN: 9798400700262. DOI: 10.1145/3573105.3575689. URL: https://doi.org/10.1145/3573105.3575689.

[11] Leonardo de Moura et al. "The Lean theorem prover (system description)". In: *International Conference on Automated Deduction*. Springer. 2015, pp. 378–388.

[12] Leonardo de Moura and Sebastian Ullrich. "The Lean 4 Theorem Prover and Programming Language". In: *Automated Deduction – CADE 28*. Ed. by André Platzer and Geoff Sut-

cliffe. Cham: Springer International Publishing, 2021, pp. 625–635. ISBN: 978-3-030-79876-5.

[13] Ulf Norell. "Towards a practical programming language based on dependent type theory". PhD thesis. SE-412 96 Göteborg, Sweden: Department of Computer Science and Engineering, Chalmers University of Technology, Sept. 2007.

[14] The Coq Development Team. *The Coq Proof Assistant, version 8.7.0*. Version 8.7.0. Oct. 2017. DOI: 10.5281/zenodo.1028037. URL: https://doi.org/10.5281/zenodo.1028037.

[15] Sebastian Ullrich and Leonardo de Moura. "'Do' Unchained: Embracing Local Imperativity in a Purely Functional Language (Functional Pearl)". In: *Proc. ACM Program. Lang.* 6.ICFP (Aug. 2022). DOI: 10.1145/3547640. URL: https://doi.org/10.1145/3547640.

[16] Sebastian Ullrich and Leonardo de Moura. "Beyond Notations: Hygienic Macro Expansion for Theorem Proving Languages". In: *CoRR* abs/2001.10490 (2020). arXiv: 2001.10490. URL: https://arxiv.org/abs/2001.10490.

[17] Sebastian Ullrich and Leonardo de Moura. "Counting Immutable Beans: Reference Counting Optimized for Purely Functional Programming [in press]". In: *31st Symposium on Implementation and Application of Functional Languages*. 31st Symposium on Implementation and Application of Functional Languages. IFL 2019 (Singapur, Singapur, Sept. 25–27, 2019). 2019.