Structures in Dependent Type Theory

Jeremy Avigad

Department of Philosophy
Department of Mathematical Sciences
Hoskinson Center for Formal Mathematics

Carnegie Mellon University

February 15, 2024

(reporting on work by the Lean development team, the Lean community, Josh Clune, Yicheng Qian, and Alex Bentkamp)

Mathematicians and formalization

In 2017, there were very few mathematicians using proof assistants.

Since then, many mathematicians have embraced Lean.

Why?

- It's a well-designed system.
- It was written with both mathematicians and computer scientists in mind.
- Documentation was written with both mathematicians and computer scientists in mind.
- Some enthusiastic mathematicians drew attention early on.
- There's an energetic online community.
- Mario Carneiro was there early on.

Mathematicians and formalization

More reasons mathematicians are comfortable with Lean:

- It has good support for classical mathematical reasoning.
- It has good support for reasoning about structures.

Systems like Isabelle and HOL Light meet the first requirement.

Systems like Coq and Agda meet the second requirement.

Both are essential.

Classical reasoning

```
noncomputable section
open Classical

def fact : ℕ → ℕ
  | 0 => 1
  | (n + 1) => (n + 1) * fact n

#eval fact 1000

def f (x : ℝ) : ℝ := if x ≤ 0 then 0 else 1

def g (x : ℝ) : ℝ := if Irrational x then 0 else 1

example : ∀ x, f x ≤ 1 := by
  intro x; simp [f]; split <;> linarith
```

Structural reasoning

Since the early twentieth century, axiomatically characterized structures have been central to mathematics.

Mathematicians now

- take products of structures,
- take powers of structures,
- take limits of structures,
- build quotients of structures,
- and lots more.

In short, they calculate with structures as easily as they calculate with numbers.

Structures have to be first-class objects in a proof assistant.

Structural reasoning

This rules out simple type theory as an attractive option.

Set theory is an alternative, but types do a lot of work:

- They allow users to overload notation and leave information implicit.
- They provide better error messages.

Clarification:

- Any foundational framework can be made to work, with enough effort.
- Mizar-style soft-typing may be an option.

Claim: any practical solution will be at least as complicated as dependent type theory.

Structural reasoning

From Sébastian Gouëzel's web page (c. 2018?):

"Out of curiosity, I have given a try to several proof assistants, i.e., computer programs on which one can formalize and check mathematical proofs, from the most basic statements (definition of real numbers, say) to the most advanced ones (hopefully including current research in a near or distant future). The first one I have managed to use efficiently is Isabelle/HOL. In addition to several facts that have been added to the main library (for instance conditional expectations), I have developed the following theories..."

Structural reasoning

"However, I have been stuck somewhat by the limitations of the underlying logic in Isabelle (lack of dependent types, making it hard for instance to define the p-adic numbers as this should be a type depending on an integer parameter p, and essentially impossible to define the Gromov-Hausdorff distance between compact metric spaces without redefining everything on metric spaces from scratch, and avoiding typeclasses). These limitations are also what makes Isabelle/HOL simple enough to provide much better automation than in any other proof assistant, but still I decided to turn to a more recent system, Lean, which is less mature, has less libraries, and less automation, but where the underlying logic (essentially the same as in Coq) is stronger (and, as far as I can see, strong enough to speak in a comfortable way about all mathematical objects I am interested in)."

Overview

I will talk about Lean's handling of structures.

- *The good:* Lean and Mathlib support an extensive network of structures.
- *The bad:* The complexity poses challenges for library development and maintenance.
- *The ugly:* It also poses challenges for automation.

Structural reasoning is one of the sources of the impressive power of modern mathematical reasoning.

With great power there must also come great responsibility.

Structures

Quiz:

- Who first gave an axiomatic characterization of a group?
- Who first defined a quotient group?
- Who first defined the notion of an ideal in a ring (and proved unique factorization of ideals)?
- Who first gave the modern definition of a Riemann surface?
- Who first gave an axiomatic characterization of a topological space?
- Who first gave an axiomatic characterization of a Hilbert space?
- Who first gave the modern definition on a measure on a space as a σ-additive function on a σ-algebra?
- Who defined the p-adic integers?

Structural language

```
def quadraticChar (α : Type) [MonoidWithZero α] (a : α) : ℤ :=
if a = 0 then 0 else if IsSquare a then 1 else -1

def legendreSym (p : ℕ) (a : ℤ) : ℤ := quadraticChar (ZMod p) a

variable {p q : ℕ} [Fact p.Prime] [Fact q.Prime]

theorem quadratic_reciprocity (hp : p ≠ 2) (hq : q ≠ 2)
    (hpq : p ≠ q) :
  legendreSym q p * legendreSym p q = (-1) ^ (p / 2 * (q / 2))
```

Structural language

```
def Padic (p : ℕ) [Fact p.Prime] :=
  CauSeq.Completion.Cauchy (padicNorm p)

def PadicInt (p : ℕ) [Fact p.Prime] :=
  { x : ℚ_[p] // ‖x‖ ≤ 1 }

variable {p : ℕ} [Fact p.Prime] {F : Polynomial ℤ_[p]}
  {a : ℤ_[p]}

theorem hensels_lemma :
    (hnorm : ‖Polynomial.eval a F‖ <
      ‖Polynomial.eval a (Polynomial.derivative F)‖ ^ 2)
  ∃ z : ℤ_[p],
    F.eval z = 0 ∧
    ‖z - a‖ < ‖F.derivative.eval a‖ ∧
    ‖F.derivative.eval z‖ = ‖F.derivative.eval a‖ ∧
    ∀ z', F.eval z' = 0 →
      ‖z' - a‖ < ‖F.derivative.eval a‖ → z' = z
```

Structural language

```
def FreeAbelianGroup : Type :=
  Additive <| Abelianization <| FreeGroup α

def IsPGroup (p : ℕ) (G : Type) [Group G] : Prop :=
  ∀ g : G, ∃ k : ℕ, g ^ p ^ k = 1

theorem IsPGroup.exists_le_sylow {P : Subgroup G}
    (hP : IsPGroup p P) :
  ∃ Q : Sylow p G, P ≤ Q
```

Structural language

```
variable {R S : Type} (K L : Type) [EuclideanDomain R]
variable [CommRing S] [IsDomain S]
variable [Field K] [Field L]
variable [Algebra R K] [IsFractionRing R K]
variable [Algebra K L] [FiniteDimensional K L] [IsSeparable K L]
variable [algRL : Algebra R L] [IsScalarTower R K L]
variable [Algebra R S] [Algebra S L]
variable [ist : IsScalarTower R S L]
variable [iic : IsIntegralClosure S R L]
variable (abv : AbsoluteValue R ℤ)

/-- The main theorem: the class group of an integral closure `S`
of `R` in a finite extension `L` of `K = Frac(R)` is finite
if there is an admissible absolute value. -/
noncomputable def fintypeOfAdmissibleOfFinite :
    Fintype (ClassGroup S) :=
  ...
```

Structural language

```
variable {α β ι : Type} {m : MeasurableSpace α}
variable [MetricSpace β] {μ : Measure α}
variable [SemilatticeSup ι] [Nonempty ι] [Countable ι]
variable  {f : ι → α → β} {g : α → β} {s : Set α}

/-- Egorov's theorem: A sequence of almost everywhere
convergent functions converges uniformly except on an
arbitrarily small set. -/
theorem tendstoUniformlyOn_of_ae_tendsto
    (hf : ∀ n, StronglyMeasurable (f n))
    (hg : StronglyMeasurable g)
    (hsm : MeasurableSet s) (hs : μ s ≠ ∞)
    (hfg : ∀ᵐ x ∂μ, x ∈ s →
      Tendsto (fun n => f n x) atTop (𝓝 (g x)))
    {ε : ℝ} (hε : 0 < ε) :
  ∃ (t : _) (_ : t ⊆ s),
    MeasurableSet t ∧
    μ t ≤ ENNReal.ofReal ε ∧
    TendstoUniformlyOn f g atTop (s \ t) :=
  ...
```

Structures in dependent type theory

```
structure Point where
  x : ℝ
  y : ℝ
  z : ℝ

def myPoint1 : Point where
  x := 2
  y := -1
  z := 4

def myPoint2 : Point := ⟨2, -1, 4⟩

#check myPoint1.x
#check myPoint1.y
#check myPoint1.z

def add (a b : Point) : Point :=
  ⟨a.x + b.x, a.y + b.y, a.z + b.z⟩
```

Structures in dependent type theory

```
structure StandardTwoSimplex where
  x : ℝ
  y : ℝ
  z : ℝ
  x_nonneg : 0 ≤ x
  y_nonneg : 0 ≤ y
  z_nonneg : 0 ≤ z
  sum_eq : x + y + z = 1

def midpoint (a b : StandardTwoSimplex) : StandardTwoSimplex
    where
  x := (a.x + b.x) / 2
  y := (a.y + b.y) / 2
  z := (a.z + b.z) / 2
  x_nonneg :=
    div_nonneg (add_nonneg a.x_nonneg b.x_nonneg) (by norm_num)
  y_nonneg := ...
  z_nonneg := ...
  sum_eq := by field_simp; linarith [a.sum_eq, b.sum_eq]
```

Structures in dependent type theory

```
structure Group where
  carrier : Type
  mul : carrier → carrier → carrier
  one : carrier
  inv : carrier → carrier
  mul_assoc : ∀ x y z : carrier,
    mul (mul x y) z = mul x (mul y z)
  mul_one : ∀ x : carrier, mul x one = x
  one_mul : ∀ x : carrier, mul one x = x
  mul_left_inv : ∀ x : carrier, mul (inv x) x = one

variable (G : Group) (g₁ g₂ : G.carrier)
```

Structures in dependent type theory

```
structure Group (α : Type) where
  mul : α → α → α
  one : α
  inv : α → α
  mul_assoc : ∀ x y z : α, mul (mul x y) z = mul x (mul y z)
  mul_one : ∀ x : α, mul x one = x
  one_mul : ∀ x : α, mul one x = x
  mul_left_inv : ∀ x : α, mul (inv x) x = one

variable {G : Type} [Group G] (g₁ g₂ : G)
```

Design specifications

Doing mathematics requires:

- defining algebraic structures and reasoning about them (groups, rings, fields, . . .)
- defining instances of structures and recognizing them as such ($\mathbb{R}$ is an ordered field, a metric space, . . .)
- overloading notation ($x + y$, "f is continuous")
- inheriting structure: every normed additive group is a metric space, which is a topological space.
- defining functions and operations on structures: we can take products, powers, limits, quotients, and so on.

Design specifications

Structure is inherited in various ways:

- Some structures extend others by adding more axioms (a commutative ring is a ring, a Hausdorff space is a topological space).
- Some structures extend others by adding more data (a module is an abelian group with a scalar multiplication, a normed field is a field with a norm).
- Some structures are defined in terms of others (every metric space is a topological space, there are various topologies on function spaces).

Defining structures and instances

We have seen how to define the group structure `Group α` on a type α.

We can define instances of `Group α` the same way we define instances of `Point` and `StandardTwoSimplex`.

```
def permGroup {α : Type} : Group (Perm α) where
  mul f g := Equiv.trans g f
  one := Equiv.refl α
  inv := Equiv.symm
  mul_assoc f g h := (Equiv.trans_assoc _ _ _).symm
  one_mul := Equiv.trans_refl
  mul_one := Equiv.refl_trans
  mul_left_inv := Equiv.self_trans_symm
```

Defining structures and instances

We are not there yet. We need:

- *Notation:* given $\mathtt{g}_1$ $\mathtt{g}_2$ `:` `Perm` α, we want to write $\mathtt{g}_1$ `*` $\mathtt{g}_2$ and ${\mathtt{g}_1}^{-1}$ for the multiplication and inverse.
- *Definitions:* we want to use defined notions like $\mathtt{g}_1$`^n` and `conj` $\mathtt{g}_1$ $\mathtt{g}_2$.
- *Theorems:* we want to apply theorems about arbitrary groups to the permutation group.

Defining structures and instances

The magic depends on three things:

1. *Logic.* A definition that makes sense in any group takes the type of the group and the group structure as arguments.

   A theorem about the elements of an arbitrary group quantifies over the type of the group and the group structure.

2. *Implicit arguments.* The arguments for the type and the structure are generally left implicit.
3. *Type class inference.*
   - Instance relations are registered with the system.
   - The system uses this information to resolve implicit arguments.

Notation

We overload notation by associating it to trivial structures.

```
class Add (α : Type u) where
  add : α → α → α

#check @Add.add
-- @Add.add : {α : Type u_1} → [self : Add α] → α → α → α

infixl:65 " + "   => Add.add

instance : Add Point where
  add := Point.add
```

Notation

```
variable (p q : Point)

#check p + q
-- p + q : Point

set_option pp.notation false
#check p + q
-- Add.add p q

set_option pp.explicit true
#check p + q
-- @Add.add Point instPointAdd p q

-- This is a slight simplification! We also have `HAdd`.
```

Classes and instances

The `class` command is a variant of the structure command that makes the structure a target for *type class inference*.

The `instance` command registers particular instances for type class inference.

We can register concrete instances ($\mathbb{R}$ is a field, the permuations of α form a group), as well as generic instances (every field is a ring, every metric space is a topological space, every normed abelian group is a metric space.)

Defining structures and instances

```
class Group (α : Type) :=
  ...

instance {α : Type} : Group (Perm α) :=
  ...

instance : Ring ℝ :=
  ...

instance {M : Type} [MetricSpace M] :
    TopologicalSpace M :=
  ...

-- Again, this is a simplification.
```

Defining structures and instances

```
#check @Add.add
-- @Add.add : {α : Type u_1} → [self : Add α] → α → α → α

#check @add_comm
-- @add_comm : ∀ {G : Type u_1} [inst : AddCommSemigroup G]
--   (a b : G), a + b = b + a

#check @abs_add
-- @abs_add : ∀ {α : Type u_1}
--    [inst : LinearOrderedAddCommGroup α] (a b : α),
--      |a + b| ≤ |a| + |b|

#check @Continuous
-- @Continuous : {α : Type u_2} → {β : Type u_1} →
--    [inst : TopologicalSpace α] →
--    [inst : TopologicalSpace β] →
--    (α → β) → Prop
```

Defining structures and instances

```
variable (f g : ℝ × ℝ → ℝ)

#check f + g
-- f + g : ℝ × ℝ → ℝ

example : f + g = g + f := by rw [add_comm]

#check Continuous f
-- Continuous f : Prop
```

Defining structures and instances

```
set_option pp.explicit true
#check Continuous f
/-
@Continuous (ℝ × ℝ) ℝ
  (@instTopologicalSpaceProd ℝ ℝ
    (@UniformSpace.toTopologicalSpace ℝ
    (@PseudoMetricSpace.toUniformSpace ℝ
    Real.pseudoMetricSpace))
    (@UniformSpace.toTopologicalSpace ℝ
    (@PseudoMetricSpace.toUniformSpace ℝ
    Real.pseudoMetricSpace)))
  (@UniformSpace.toTopologicalSpace ℝ
    (@PseudoMetricSpace.toUniformSpace ℝ
    Real.pseudoMetricSpace)) f : Prop
-/
```

Defining a hierarchy of structures

Currently, Mathlib has roughly:

- 1,300 classes
- 22,110 instances.

I will pause here to show you:

- some graphs
- how to look up the (direct) instances of a class in the Mathlib documentation
- how to look up the classes that an object is an instance of.

Defining a hierarchy of structures

What are all these classes?

- Notation (`Add`, `Mul`, `Inv`, `Norm`, ...)
- Algebraic structures (`Group`, `OrderedRing`, `Lattice`, `Module`, ...)
- Computation and bookkeeping: `Inhabited`, `Decidable`
- Mixins and add-ons: `LeftDistribClass`, `Nontrivial`
- Unexpected generalizations: `GroupWithZero`, `DivInvMonoid`

Defining a hierarchy of structures

```
class DivisionSemiring (α : Type*) extends Semiring α,
    GroupWithZero α

class DivisionRing (K : Type u) extends Ring K, DivInvMonoid K,
    Nontrivial K, RatCast K

class Semifield (α : Type*) extends CommSemiring α,
    DivisionSemiring α, CommGroupWithZero α

class Field (K : Type u) extends CommRing K, DivisionRing K
```

To bundle or not to bundle?

A *group* consists of a carrier type and a structure on that type.

We have seen that we can represent that as one object or two.

Choices like this come up often:

- A monoid morphism is a function that preserves multiplication and 1.
- A subgroup is a subset of the carrier closed under the group operations.

To bundle or not to bundle?

```
variable (G H : Type) [Monoid G] [Monoid H]

structure isMonoidHom (f : G → H) : Prop where
  map_one : f 1 = 1
  map_mul : ∀ g g', f (g * g') = f g * f g'

structure MonoidHom : Type where
  toFun : G → H
  map_one : toFun 1 = 1
  map_mul : ∀ g g', toFun (g * g') = toFun g * toFun g'

structure Subgroup (G : Type) [Group G] where
  carrier : Set G
  mul_mem {a b} : a ∈ carrier → b ∈ carrier →
    a * b ∈ carrier
  one_mem : (1 : G) ∈ carrier
  inv_mem {x} : x ∈ carrier → x⁻¹ ∈ carrier
```

To bundle or not to bundle?

The bundled and unbundled approaches each have advantages and drawbacks.

Mathlib has ways of handling subobjects and morphisms that tries to get the best of both worlds.

You can read about it in Chapter 7 of *Mathematics in Lean*.

See also Anne Baanen, "Use and abuse of instance parameters in the Lean mathematical library."

Diamond problems

Consider the following facts:

- The product of metric spaces is a metric space.
- The product of topological spaces is a topological space.
- Every metric space is a topological space.

Suppose M_1 and M_2 are metric spaces.

$M_1 \times M_2$ can be viewed as a topological space in two ways:

- A product of the induced topological spaces.
- The topological space induced by the product of the metric spaces.

Fortunately, they come out the same.

Diamond problems

Why diamonds are problematic:

- The multiple pathways slow down searches.
- The results may not be the same (an ambiguity in the mathematics).
- The results may be *provably* the same, but not syntactically (definitionally) the same.

Here's how you know things have gone wrong:

```
tactic 'apply' failed, failed to unify
  Continuous f
with
  Continuous f
```

Diamond problems

Diamond problems come up surprisingly often.

Mathematics in Lean explains how to resolve them, and the community has gotten good at it.

See also Affeldt et al, "Competing inheritance paths in dependent type theory" and Wieser, "Multiple-inheritance hazards in dependently-typed algebraic hierarchies."

A philosophical question:

- Mathematicians are good at inferring canonical structure.
- A priori, there is no guarantee that our conventions yield coherent assignments.
- Why don't we get in trouble more often?

Automation

Current automation for Lean:

- lots of small-scale tactics for doing useful things (casing on data, doing calculations, logical manipulations)
- domain specific automation, `linarith`, `ring`, `omega`, `monotonicity`, `FunProp`
- `simp`, equational reasoning and conditional simplification
- `Aesop`, a tableaux reasoner like Isabelle's `auto`.

We don't have anything like Isabelle's *sledgehammer*.

Automation

Isabelle's sledgehammer:

- uses premise selection to choose a manageable set of relevant facts from the library
- exports problems from Isabelle to external ATPs and SMT solvers
- uses the results of the external tools to construct formally verified proofs.

Two key components:

- a monomorphization / translation procedure
- metis, a proof-producing resolution procedure.

What will these look like in Lean?

Automation

Targets for automation:

- equational reasoning
- propositional reasoning
- domain-specific decision procedures (linear arithmetic, linear integers arithmetic)
- first-order reasoning
- higher-order reasoning
- dependent types
- structural reasoning

Establishing a goal in Lean generally requires a mixture of these.

Automation

Josh Clune, Yicheng Qian, and Alex Bentkamp are working on *Duper*:

- the Lean analogue of Metis
- can be used as internal automation and for proof reconstruction
- can handle some higher-order reasoning and features of dependent type theory.

Yicheng Qian has been working on *Lean-Auto*:

- a tool that can translate Lean goals to TPTP and SMT format
- it can also do preprocessing for Duper
- it has to deal with specific features of dependent type theory.

Their talks from *Lean Together 2024* are online.

Automation

Duper's "comfort zone" is first-order reasoning.

But it has mechanisms to handle:

- type inhabitation
- variables over types
- variables over type universes
- higher-order unification

The last is expensive, but is needed e.g. to use identities like this:

$$\sum_{x \in A} x^2 + 2x = \sum_{x \in A} x^2 + 2 \sum_{x \in A} x.$$

Automation

Lean-Auto provides preprocessing for Duper as well as external tools.

Given a Lean expression e, Lean-Auto produces a simpler expression e', an assignment σ, and a proof that e is equal to evaluating e' under σ.

Duper or external automation can then be used to prove the simpler expression.

Automation

Things that Lean-Auto does:

- Canonicalizes expressions, so e.g. two definitionally equal expressions (like $s \leq t$) are represented by the same term.
- Iteratively instantiates theorems $\forall \alpha \ \ldots$, where α ranges over type variables. (Monomorphization)
- It instantiates type classes.
- Similarly instantiates other variables that types depend on.
- It moves everything to one universe.
- It does other preprocessing.

Automation

To use external automation, we can use premise selection, Lean-Auto, and proof reconstruction.

For internal Lean automation and proof reconstruction we need:

- equational reasoning (Duper, simp)
- propositional reasoning (Duper)
- domain-specific decision procedures (linarith, omega)
- first-order reasoning (Duper)
- higher-order reasoning (Lean-Auto or Duper)
- dependent types (Lean-Auto or Duper)
- structural reasoning (Lean-Auto)

Conclusions

- Structural reasoning is one of the most salient and powerful features of modern mathematics.
- Any viable proof assistant for mathematics has to support it.
- It's not easy.
- Lean and Mathlib do pretty well.
- We are not out of the woods.
  - The complexity poses challenges for library development and maintenance.
  - It also poses challenges for automation.