Recent development in Lean and its analysis

Johannes Hölzl
(Matryoshka @ VU Amsterdam)

Linear Algebra in Isabelle/HOL
Universidad de La Rioja

THEOREM PROVER

Lean and its analysis

Not about Isabelle's linear algebra, but about Lean.

- interesting concepts / syntax / tools / ...
- some of these would fit for Isabelle
- thinking outside the boundary of HOL
- convince people to use Lean!

Outline

- What is Lean
  - Lean Architecture
  - Dependent Types (Uniform Syntax)
  - Some Syntactic Sugar
  - ...
- Library
  - Topology: Uniform spaces and Reals
  - Summation operator
  - Measure theory
  - Recently: cardinals

What is Lean

Lean architecture

User Interface (e.g. VS Code or Emacs)

Elaborator

- syntax sugar
- type inference + classes
- equation compiler

VM

- execute meta-expressions
- tactics

Kernel

- expressions + type checker
- declarations
- inductive + quotient types

Dependent Types extend HOL

Types are terms!

- $\mathbb{N} : \mathtt{Type}_0$

Dependent Types extend HOL

Types are terms!

- $\mathbb{N} : \texttt{Type}_0$
- $\texttt{Type}_u : \texttt{Type}_{u+1}$
  HOL happens in $\texttt{Type}_0$

Dependent Types extend HOL

Types are terms!

- $\mathbb{N} : \texttt{Type}_0$
- $\texttt{Type}_u : \texttt{Type}_{u+1}$
  HOL happens in $\texttt{Type}_0$
- $\alpha : \texttt{Type}_u \implies \texttt{list}\ \alpha : \texttt{Type}_u$,
  i.e. $\texttt{list} : \texttt{Type}_u \to \texttt{Type}_u$

Dependent Types extend HOL

Types are terms!

- $\mathbb{N} : \texttt{Type}_0$
- $\texttt{Type}_u : \texttt{Type}_{u+1}$
  HOL happens in $\texttt{Type}_0$
- $\alpha : \texttt{Type}_u \implies \texttt{list}\ \alpha : \texttt{Type}_u$,
  i.e. $\texttt{list} : \texttt{Type}_u \to \texttt{Type}_u$
- $\texttt{vec} : \texttt{Type}_u \to \mathbb{N} \to \texttt{Type}_u$

Dependent Types extend HOL

Types are terms!

- $\mathbb{N} : \mathtt{Type}_0$
- $\mathtt{Type}_u : \mathtt{Type}_{u+1}$
  HOL happens in $\mathtt{Type}_0$
- $\alpha : \mathtt{Type}_u \implies \mathtt{list}\ \alpha : \mathtt{Type}_u$,
  i.e. $\mathtt{list} : \mathtt{Type}_u \to \mathtt{Type}_u$
- $\mathtt{vec} : \mathtt{Type}_u \to \mathbb{N} \to \mathtt{Type}_u$
- Also: types can be empty!

Dependent Types extend HOL

Proofs are terms!

- $\texttt{Prop} : \texttt{Type}_0$
  — type universe of propositions (i.e. $\simeq$ `bool`)

Dependent Types extend HOL

Proofs are terms!

- $\mathtt{Prop} : \mathtt{Type}_0$
  — type universe of propositions (i.e. $\simeq$ `bool`)
- $\mathtt{true}, \mathtt{false}, \ldots : \mathtt{Prop}$
  — propositions are types

Dependent Types extend HOL

Proofs are terms!

- $\texttt{Prop} : \texttt{Type}_0$
  — type universe of propositions (i.e. $\simeq$ $\texttt{bool}$)
- $\texttt{true}, \texttt{false}, \ldots : \texttt{Prop}$
  — propositions are types
- $\texttt{trueI} : \texttt{true}$
  — proofs are the elements of propositions

Dependent Types extend HOL

Proofs are terms!

- $\texttt{Prop} : \texttt{Type}_0$
  — type universe of propositions (i.e. $\simeq$ $\texttt{bool}$)
- $\texttt{true}, \texttt{false}, \ldots : \texttt{Prop}$
  — propositions are types
- $\texttt{trueI} : \texttt{true}$
  — proofs are the elements of propositions
- $\texttt{false}$ is empty!
  — $\forall \alpha, \texttt{false} \to \alpha$

Dependent Types in Lean

Lean is now:

- a little bit of outer syntax
- dependent type language + (a lot of) syntactic sugar
- one language to express: terms, types, proofs

```
def double (a : ℤ) : ℤ :=
a + a

lemma double_0 : double 0 = 0 :=
add_zero 0
```

Ex: recursion for types, fun, & proofs

```
-- Type ‘vec‘
def vec (α : Type) : ℕ → Type
| 0       := unit
| (n + 1) := α × vec n
```

Ex: recursion for types, fun, & proofs

```
-- Type ‘vec‘
def vec (α : Type) : ℕ → Type
| 0       := unit
| (n + 1) := α × vec n

-- Function ‘map‘
def map (α β : Type) (f : α → β) :
  Π (n : ℕ), vec α n → vec β n
| 0       ()     := ()
| (n + 1) (a, v) := (f a, map n v)
```

Ex: recursion for types, fun, & proofs

```
-- Type ‘vec‘
def vec (α : Type) : ℕ → Type
| 0       := unit
| (n + 1) := α × vec n

-- Function ‘map‘
def map (α β : Type) (f : α → β) :
  Π (n : ℕ), vec α n → vec β n
| 0       ()     := ()
| (n + 1) (a, v) := (f a, map n v)

-- Theorem ‘map_id‘
lemma map_id (α : Type) :
  ∀ (n : ℕ) (v : vec α n), map α α id n v = v
| 0       ()     := rfl
| (n + 1) (a, v) := by simp [map, map_id n v]
```

Nice case analysis for proofs

Equation compiler allows a precise case analysis:

```
lemma ex : ∀i (s : set ℕ),
  (∃n∈s, f n = i) ∨ s = ∅ → P i s
| _ s (or.inl ⟨n, hns, rfl⟩) :=
  show P (f n) s, from sorry
| i _ (or.inr rfl) :=
  show P i ∅, from sorry
```

Some syntactic sugar

- Anonymous constructor:
  $\langle$`0, ()`$\rangle$ `: Σn, vec ℤ n,`
  $\langle$`2, by simp`$\rangle$ `: ∃n, n = 1 + 1`

Some syntactic sugar

- Anonymous constructor:
  `⟨0, ()⟩ : Σn, vec ℤ n,`
  `⟨2, by simp⟩ : ∃n, n = 1 + 1`
- Special "dot"-syntax: `xs.map f` where

Some syntactic sugar

- Anonymous constructor:
  `⟨0, ()⟩ : Σn, vec ℤ n,`
  `⟨2, by simp⟩ : ∃n, n = 1 + 1`
- Special "dot"-syntax: `xs.map f` where
  - `xs : list` α

Some syntactic sugar

- Anonymous constructor:
  `⟨0, ()⟩ : Σn, vec ℤ n,`
  `⟨2, by simp⟩ : ∃n, n = 1 + 1`
- Special "dot"-syntax: `xs.map f` where
  - `xs : list` α
  - `list.map : (`$\alpha \to \beta$`)` $\to$ `list` $\alpha \to$ `list` β

Some syntactic sugar

- Anonymous constructor:
  `⟨0, ()⟩ : Σn, vec ℤ n,`
  `⟨2, by simp⟩ : ∃n, n = 1 + 1`
- Special "dot"-syntax: `xs.map f` where
  - `xs : list` α
  - `list.map : (`$\alpha \to \beta$`)` $\to$ `list` $\alpha \to$ `list` β
  - Result: `list.map f xs`

Some syntactic sugar

- Anonymous constructor:
  `⟨0, ()⟩ : Σn, vec ℤ n,`
  `⟨2, by simp⟩ : ∃n, n = 1 + 1`
- Special "dot"-syntax: `xs.map f` where
  - `xs : list` α
  - `list.map : (`$\alpha \to \beta$`)` $\to$ `list` $\alpha \to$ `list` β
  - Result: `list.map f xs`
  - Replaces `map_` < `type` >, `set_` < `type` >, ...

Some syntactic sugar

- Anonymous constructor:
  `⟨0, ()⟩ : Σn, vec ℤ n,`
  `⟨2, by simp⟩ : ∃n, n = 1 + 1`

- Special "dot"-syntax: `xs.map f` where
  - `xs : list` α
  - `list.map : (`$\alpha \to \beta$`)` $\to$ `list` $\alpha \to$ `list` β
  - Result: `list.map f xs`
  - Replaces `map_` $<$ `type` $>$, `set_` $<$ `type` $>$, $\ldots$

- Haskell $:
  $f\ a\ \$\ g\ b\ \$\ h\ x$ instead of $f\ a\ (g\ b\ (h\ x))$

Library

Basic algebraic and order hierarchy

Lean follows mostly Isabelles algebraic and order hierarchy

- (partial) orders, (complete) lattices, ...
- (commutative) semigroups, monoids, groups, rings, and finally fields
- Start separating type classes containing constants and pure predicates.
  This makes also a difference in Isabelle

```
class module
  (α : inout Type u) (β : Type v) [inout ring α]
  extends has_scalar α β, add_comm_group β :=
...
```

Topology

- Filter library

Topology

- Filter library
- Hierarchy follows Isabelle

Topology

- Filter library
- Hierarchy follows Isabelle
- Continuity is unbounded: `continuous f`

Topology

- Filter library
- Hierarchy follows Isabelle
- Continuity is unbounded: `continuous f`
- Operations on the structure itself:
  `complete_lattice(topological_space` α`)`
  $\Rightarrow$ constructions (nearly) for free

Reals

- Uniform completion of $\mathbb{Q}$

Reals

- Uniform completion of $\mathbb{Q}$
- Uniform spaces generalize metric spaces, but do not require $\mathbb{R}$
  Rely heavily on filters

Reals

- Uniform completion of $\mathbb{Q}$
- Uniform spaces generalize metric spaces, but do not require $\mathbb{R}$
  Rely heavily on filters
- Hope: do uniform completion generally and instantiate for $\mathbb{R}$
  Did *not* work out, still requires a lot of work

Reals

- Uniform completion of $\mathbb{Q}$
- Uniform spaces generalize metric spaces, but do not require $\mathbb{R}$
  Rely heavily on filters
- Hope: do uniform completion generally and instantiate for $\mathbb{R}$
  Did *not* work out, still requires a lot of work
- Finally: metric and order complete field

Measure theory

Finally the freedom to do them right!

```
class measurable_space (α : Type u) := ...

class measure_space
  (α : Type u) [measurable_space α] :=
(measure_of : Πs, is_measurable s → ennreal)
...
```

- with complete lattice structure, map, comap, ...
- currently up to the Lebesgue measure

Cardinals

- $\alpha \simeq \beta := \exists f : \alpha \to \beta, f \text{ bijective}$

Cardinals

- $\alpha \simeq \beta := \exists f : \alpha \to \beta, f \text{ bijective}$
- $\simeq$ is an equivalence relation on types!

Cardinals

- $\alpha \simeq \beta := \exists f : \alpha \to \beta, f \text{ bijective}$
- $\simeq$ is an equivalence relation on types!
- $\texttt{cardinals}_u : \texttt{Type}_{u+1} := \texttt{Type}_{u/\simeq}$

Cardinals

- $\alpha \simeq \beta := \exists f : \alpha \to \beta, f$ bijective
- $\simeq$ is an equivalence relation on types!
- $\texttt{cardinals}_u : \texttt{Type}_{u+1} := \texttt{Type}_{u/\simeq}$
- unbounded cardinals: closed under $\mathcal{P}$

Cardinals

- $\alpha \simeq \beta := \exists f : \alpha \to \beta, f \text{ bijective}$
- $\simeq$ is an equivalence relation on types!
- $\texttt{cardinals}_u : \texttt{Type}_{u+1} := \texttt{Type}_{u/\simeq}$
- unbounded cardinals: closed under $\mathcal{P}$
- semiring and total order (no wellorder yet)

Cardinals

- $\alpha \simeq \beta := \exists f : \alpha \to \beta, f \text{ bijective}$
- $\simeq$ is an equivalence relation on types!
- $\texttt{cardinals}_u : \texttt{Type}_{u+1} := \texttt{Type}_{u/\simeq}$
- unbounded cardinals: closed under $\mathcal{P}$
- semiring and total order (no wellorder yet)
- Example application:
  should allow most BNF constructions

Conclusion

- type constructions are everywhere in DTT

Conclusion

- type constructions are everywhere in DTT
- some constructions can be also done in Isabelle

Conclusion

- type constructions are everywhere in DTT
- some constructions can be also done in Isabelle
- ...

Thanks for listening