Lean and its Type System

Fabian Schaub

June 13, 2022

Outline

- About Lean
- Programming Language
- Type System
- Theorem Proving

About Lean

LEAN THEOREM PROVER

- Launched 2013 by Leonardo de Moura @ Microsoft Research
- Pure functional progamming language
- Latest version: Lean 4

The Language

Functions

- Pure functions

```
def name := "Anshalm"
def greet (n : String) : String := s!"Hello, {n}!"
```

The Language

Functions

- Pure functions

```
def name := "Anshalm"
def greet (n : String) : String := s!"Hello, {n}!"
```

- Monadic expressions and Do-Notation

```
def doGreet : IO Unit :=
  pure (greet name) >>= λ g => IO.println g

def main : IO Unit := do
  let g ← pure (greet name)
  IO.println g
```

The Language

Functions

- Pure functions

```
def name := "Anshalm"
def greet (n : String) : String := s!"Hello, {n}!"
```

- Monadic expressions and Do-Notation

```
def doGreet : IO Unit :=
  pure (greet name) >>= λ g => IO.println g

def main : IO Unit := do
  let g ← pure (greet name)
  IO.println g
```

- Evaluating Expressions

```
#check greet name -- String
#eval greet name  -- "Hello, Anshalm!"
```

- Recursive functions need to be terminating
  - Show termination by hand

The Language

Recursive Functions

- Recursive functions need to be terminating
  - Show termination by hand
- Use partial recursive function
  - Type has to be non-empty

The Language

Recursive Functions Cont.

- Recursive Functions

```
-- fails with 'fail to show termination'
def loop1 (a : Nat) : Nat :=
  match a with
  | 0 => a
  | _ => loop1 (a - 1)
```

The Language

Recursive Functions Cont.

- Recursive Functions

```
-- fails with 'fail to show termination'
def loop1 (a : Nat) : Nat :=
  match a with
  | 0 => a
  | _ => loop1 (a - 1)


-- define a partial function
partial def loop2 (cond : Nat -> Bool) (a : Nat) :
    Nat :=
  if cond a then a else loop2 cond (a - 1)
```

- Data Types

```
inductive Weekday where
  | sunday    : Weekday
  | monday    : Weekday
  | ...

structure Point (α : Type u) where
  x : α
  y : α
```

The Language
Inductive Data Types

- Inductive Data Types

```
inductive Tree (α : Type u) where
  | node : Tree α -> α -> Tree α -> Tree α
  | leaf : Tree α
```

The Language

Type Classes

- Type Classes

```
class Add (a : Type) where
  add : a -> a -> a
instance : Add Nat where
  add x y := x + y

instance [Add α] : Add (Maybe α) where
  add x y :=
    match x with
    | Maybe.nothing => Maybe.nothing
    | Maybe.just a  =>
      match y with
      | Maybe.nothing => Maybe.nothing
      | Maybe.just b  => Maybe.just (a + b)

def double [Add α] (a : α) : α :=
  Add.add a a
```

Demo

Programming

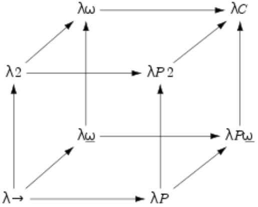
Type System

Leans Type System builds on the Calculus of Constructions (λ_C) with Inductive Types (Calculus of Inductive Constructions).

Type System

Leans Type System builds on the Calculus of Constructions (λ_C) with Inductive Types (Calculus of Inductive Constructions).

- $\lambda_{\to}$ - Simply Typed LC
- λ_2 - Polymorphism
- λ_P - Dependent Types
- $\lambda_{\underline{\omega}}$ - (inductive) Type Constructors



Type System

λ_2 - Polymorphism

$$\mathit{compose} \ : \ \forall\alpha.\forall\beta.\forall\gamma.(\beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma$$

$$\equiv$$

```
def compose (α β γ : Type) (g : β → γ) (f : α → β) (x
    : α) : γ :=
  g (f x)
```

Type System

λ_P - Dependent Types

$$foo : (\Pi a : String \; . \; B)$$

$$\equiv$$

```
def foo (s : String) : Type := match s with
  | "String" => String
  | "Nat" => Nat
  | _ => Bool
```

Type System

$\lambda_{\underline{\omega}}$ - (inductive) Type Constructors

```
inductive Maybe (α : Type u) where
  | just : α -> Maybe α
  | nothing : Maybe α

def isJust (α : Type u) (a : Maybe α) : Bool :=
  match a with
  | Maybe.just _  => true
  | _             => false
```

Type System

Universes

- Infinite hierarchy of universes
- Each type therein is denoted by `Type (u : Nat)`
- `Type` is syntactic sugar for `Type 0`

Type System
Universes

- Infinite hierarchy of universes
- Each type therein is denoted by `Type (u : Nat)`
- `Type` is syntactic sugar for `Type 0`
- `Int Nat Bool ... : Type`

Type System

Universes

- Infinite hierarchy of universes
- Each type therein is denoted by `Type (u : Nat)`
- `Type` is syntactic sugar for `Type 0`
- `Int Nat Bool ... : Type`
- A “special” type: `Prop : Type`

Theorem Proving

- Propositions are encoded with `Prop : Type`
- `Prop` is closed under the arrow constructor

Theorem Proving

- Propositions are encoded with `Prop : Type`
- `Prop` is closed under the arrow constructor

```
-- the Proposition Type
variable (a b c : Prop)

-- useful Type Constructors
#check a ∧ b -- Prop
#check a ∨ b -- Prop
```

Theorem Proving

Proof a proposition by finding a suitable term.

```
variable {p : Prop}
variable {q : Prop}

-- proof a theorem by providing a term of its type
theorem t1 : p → q → p := fun hp : p => fun hq : q => hp
```

Theorem Proving

Proof a proposition by finding a suitable term.

```
variable {p : Prop}
variable {q : Prop}

-- proof a theorem by providing a term of its type
theorem t1 : p → q → p := fun hp : p => fun hq : q => hp
```

NOTE: `theorem` is logically equivalent to `def`.

Theorem Proving

With `axiom name : proposition` we define axioms.

Theorem Proving

With `axiom name : proposition` we define axioms.

```
variable {p : Prop}
variable {q : Prop}

theorem t1 (hp : p) (hq : q) : p := hp
```

Theorem Proving

With `axiom name : proposition` we define axioms.

```
variable {p : Prop}
variable {q : Prop}

theorem t1 (hp : p) (hq : q) : p := hp


-- define an axiom
axiom myaxiom : p
```

Theorem Proving

With `axiom name : proposition` we define axioms.

```
variable {p : Prop}
variable {q : Prop}

theorem t1 (hp : p) (hq : q) : p := hp


-- define an axiom
axiom myaxiom : p


-- Modus Ponens corresponds to β-reduction
theorem t2 : q → p := t1 myaxiom
```

Theorem Proving

We define further theorems.

```
variable {p : Prop}
variable {q : Prop}

-- define an and introduction rule
theorem and_intro : p -> q -> p ∧ q :=
  fun hp hq => ⟨hp, hq⟩
```

Theorem Proving

We define further theorems.

```
variable {p : Prop}
variable {q : Prop}

-- define an and introduction rule
theorem and_intro : p -> q -> p ∧ q :=
  fun hp hq => ⟨hp, hq⟩


-- define and symmetry rule
theorem and_symmetry (h : p ∧ q) : q ∧ p := and_intro
    h.right h.left
```

Additionally, Lean has a tactics mode

```
theorem program_mode (p q : Prop) (hp : p) (hq : q) : p ∧
    q :=
  And.intro hp hq
```

Additionally, Lean has a tactics mode

```
theorem program_mode (p q : Prop) (hp : p) (hq : q) : p ∧
    q :=
  And.intro hp hq
```

```
theorem tactics_mode (p q : Prop) (hp : p) (hq : q) : p ∧
    q := by
  apply And.intro
  exact hp
  exact hq
```

Demo

Proving (Part 1)

Theorem Proving

Implementing Proposition Constructors

Propositions are implemented using inductive types.

```
inductive False : Prop

inductive True : Prop where
  | intro : True

inductive And (a b : Prop) : Prop where
  | intro : a → b → And a b

inductive Or (a b : Prop) : Prop where
  | inl : a → Or a b
  | inr : b → Or a b

inductive Exists {α : Type u} (q : α → Prop) : Prop where
  | intro : ∀ (a : α), q a → Exists q
  -- ∃ x : α, p is syntactic sugar for
  -- Exists (fun x : α => p)
```

Theorem Proving

Induction

Inductive types have an inductive type definition:

`Type.rec` and `Type.recOn`

Theorem Proving

Induction

Inductive types have an inductive type definition:

`Type.rec` and `Type.recOn`

```
inductive Nat where
  | zero : Nat
  | succ : Nat → Nat
```

Theorem Proving

Induction

Inductive types have an inductive type definition:

`Type.rec` and `Type.recOn`

```
inductive Nat where
  | zero : Nat
  | succ : Nat → Nat

#check @Nat.rec
  : {motive : Nat → Sort u}
  → motive Nat.zero
  → ((n : Nat) → motive n → motive (Nat.succ n))
  → (t : Nat) → motive t
```

Theorem Proving

Induction

Inductive types have an inductive type definition:

`Type.rec` and `Type.recOn`

```
inductive Nat where
  | zero : Nat
  | succ : Nat → Nat

#check @Nat.rec
  : {motive : Nat → Sort u}
  → motive Nat.zero
  → ((n : Nat) → motive n → motive (Nat.succ n))
  → (t : Nat) → motive t

#check @Nat.recOn
  : {motive : Nat → Sort u}
  → (t : Nat)
  → motive Nat.zero
  → ((n : Nat) → motive n → motive (Nat.succ n))
  → motive t
```

Demo

Proving (Part 2)

References I

Lean Manual.
https://leanprover.github.io/lean4/doc/.
[Online; accessed 13-Jun-2022].

Theorem Proving in Lean 4.
https://leanprover.github.io/theorem_proving_in_lean4/.
[Online; accessed 13-Jun-2022].

Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer.
The lean theorem prover (system description).
In *International Conference on Automated Deduction*, pages 378–388. Springer, 2015.

References II

Leonardo de Moura and Sebastian Ullrich.
The lean 4 theorem prover and programming language.
In *International Conference on Automated Deduction*, pages 625–635. Springer, 2021.

Thank you for your attention!