The Lean Proof Assistant

CS242
Lecture 15
Review

• Dependent types are a foundation for mathematics
  • And typed programming

• A single formalism for defining programs, proofs, and proof rules
  • And ensuring they are used in a consistent way

• Relies on constructive interpretations of mathematics
  • We must construct (compute) evidence for every assertion
  • Constructive proofs exclude proofs by contradiction
Once More, From the Top …

• Today we will look at Lean (version 3)
  • The proof assistant you will use in a homework assignment …

• Illustrate basic features with examples

• Focus on using Lean for proofs
  • Not exploring new type theory
Basics

Type assertions are written `e : t`, meaning expression e has type t

Examples:

```
class constant n : nat
constant f : nat -> nat
```

The `#check` command prints out information about a name

- Useful for debugging

```
#check n
#check f
#check f n
```
Browser-Based Lean

• There is a nice WebAssembly implementation of Lean
  • Simply type expressions into the browser and see the results
  • Makes it easy to experiment

https://leanprover-community.github.io/lean-web-editor/
Recall: Programs as Proofs

\[ A \vdash e_1 : t \rightarrow t' \]
\[ A \vdash e_2 : t \]
\[ A \vdash e_1 e_2 : t' \]  \hspace{1cm} \text{[App]}

\[ A, x : t \vdash e : t' \]
\[ A \vdash \lambda x. e : t \rightarrow t' \]  \hspace{1cm} \text{[Abs]}

From a proof of \( t \rightarrow t' \) and a proof of \( t \), we can prove \( t' \).

If assuming \( t \) we can prove \( t' \), then we can prove \( t \rightarrow t' \).
Function Definitions

• Lambda calculus (or implication) is built-in to Lean

• Two equivalent definitions of a function:

```lean
def app (g : nat → nat) (x : nat) : nat := g x
def app2 : (nat → nat) → nat → nat := \lam g x, g x
```
Notes

def app (g: nat -> nat) (x:nat) : nat := g x
def app2 : (nat -> nat) -> nat -> nat := \lam g x, g x

• \lam is ascii for λ
  • Lean takes unicode seriously!

• Note λ’s can have multiple variables (no need to repeat λ)
• The punctuation is different from other languages
  • Definition uses := instead of =
  • Write \lambda x, e not \lambda x. E
  • A list of variables is separated by spaces, not commas
    • Parentheses are often needed if variables are given types (c.f., the arguments to app)
  • Types can often be omitted, but not always
    • Lean has type inference, but still need enough types for Lean to figure out all the types
Polymorphic Functions

```plaintext
def polyapp (α : Type) (g: α -> α) (x:α) : α := g x
def polyapp2 : Π α : Type, (α -> α) -> α -> α := λ t g x, g x
def polyapp3 : ∀ α : Type, (α -> α) -> α -> α := λ t g x, g x
```

• These polymorphic versions take a type argument
  • And it is a dependent type – the type of the function depends on the type argument!
  • Which is why we use Π (or ∀, they are synonyms)

• Unicode: \Pi is Π, \forall is ∀, \a is α
Propositions as Types

A theorem:

\[
\begin{align*}
\text{constants } p \ q : \text{Prop} \\
\text{theorem } t1 : p \to q \to p := \lambda \ hp : p, \ \lambda \ hq : q, \ hp
\end{align*}
\]

• But \( \text{Prop} = \text{Type} \)
• And \( \text{theorem} = \text{def}! \)
• Just alternative syntax to emphasize proofs instead of computation
And More Options

• We could also write this proof

```lean
theorem t2 : p → q → p :=
  assume hp : p,
  assume hq : q,
  hp
```

• This means exactly the same thing
• `assume` is just longhand for `λ`
The Polymorphic Version

• We could also write this proof so it works for any \( p \) and \( q \)

\[
\text{theorem t3 (p,q: Prop) : p \rightarrow q \rightarrow p :=}
\]
\[
\text{assume hp : p,}
\]
\[
\text{assume hq : q,}
\]
\[
\text{hp}
\]

Alex Aiken      CS 242     Lecture 15
Conjunction: And Introduction

A few proofs of $p \rightarrow q \rightarrow p \land q$

lemma a1 (hp : p) (hq : q) : p \land q := and.intro hp hq

or

lemma a2 : p \rightarrow q \rightarrow p \land q := \lambda hp: p, \lambda hq: q, and.intro hp hq

or

lemma a3 : p \rightarrow q \rightarrow p \land q :=
    assume hp: p,
    assume hq: q,
    and.intro hp hq

or

lemma a4 (hp : p) (hq : q) : p \land q := < hp, hq >

Note: lemma is another synonym for def, the angle brackets are special syntax for and.intro
Conjunction: And Elimination

Proofs of $p \land q \rightarrow q \land p$

lemma a5 (hpq: p \land q) : q \land p := and.intro (and.right hpq) (and.left hpq)

lemma a6 (hpq: p \land q) : q \land p := and.intro hpq.right hpq.left

lemma a7 (hpq: p \land q) : q \land p := ⟨hpq.right, hpq.left⟩
Disjunction: Or Introduction

Proofs of $p \rightarrow p \lor q$ and $q \rightarrow p \lor q$

lemma o1 (hp : p) : p ∨ q := or.intro_left q hp

lemma o2 : q → p ∨ q :=
    assume hq: q,
    or.intro_right p hq
Disjunction: Or Elimination

Proofs of $p \lor q \rightarrow q \lor p$

lemma o3 (h : p \lor q) : q \lor p :=
or.elim h
  (assume hp : p,
or.intro_right q hp)
  (assume hq : q,
or.intro_left p hq)

or.elim does a case analysis
Specifically, or.elim is a function taking three arguments:

an object of type $p \lor q$
a function of type $p \rightarrow r$
a function of type $q \rightarrow r$

In this example $r = q \lor p$
Show: Making the Conclusion Explicit

lemma o3 (h : p ∨ q) : q ∨ p :=
or.elim h
  (assume hp : p,
   show q ∨ p,
   from or.intro_right q hp)
  (assume hq : q,
   show q ∨ p,
   from or.intro_left p hq)

• show allows the user to state the goal
  • The proposition (type) we are trying to prove
• Helpful for making proofs clearer
• And detecting bugs in the proof earlier
Structuring Longer Proofs

lemma a8 (h : p ∧ q) : q ∧ p :=
  have hp : p, from and.left h,
  have hq : q, from and.right h,
  show q ∧ p, from and.intro hq hp

have h from t in e
is equivalent to
(λh.e) t

Recall (λh.e) t is also equivalent to
let h = t in e

Useful for structuring longer arguments in a series of steps
A More Complex Lemma

\[(p \rightarrow q) \rightarrow (p \rightarrow r) \rightarrow (p \rightarrow q \land r)\]

lemma imp (f1: p -> q) (f2: p -> r) (x:p) : q \land r :=
  have hq: q, from f1 x,
  have hr: r, from f2 x,
  show q \land r, from \langle hq, hr \rangle
Quantifiers

• We’ve already seem examples of universal quantifiers

• Recall
  \[
  \text{def polyapp } (\alpha : \text{Type}) \ (g : \alpha \rightarrow \alpha) \ (x : \alpha) : \alpha := g \ x
  \]
  \[
  \text{def polyapp2 } : \Pi \alpha : \text{Type}, \ (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha := \lambda \ t \ g \ x, \ g \ x
  \]
  \[
  \text{def polyapp3 } : \forall \alpha : \text{Type}, \ (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha := \lambda t \ g \ x, \ g \ x
  \]

If we define polymorphic functions, we are carrying out universal proofs.

The intro and elimination of universal quantifiers is implicit in polymorphic typechecking.

A very common case, though there are times we want explicit \(\forall\)-intro and \(\forall\)-elim.
Existential Quantifier Elimination

Eliminating an existential quantifier from $h: \exists x: t, p x$ has the form

```plaintext
exists.elim h
   (assume y : t,
    assume z : p y,
    e)
```
Existential Quantifier Introduction

Consider a proposition of the form $E(p)$

The `exists.intro` $p \ E(p) = \exists \ x. \ E(x)$

We replace the subexpression $p$ by the existentially bound variable

- Not entirely trivial, as $p$ could be a complex expression that the system needs to search for in $E(p)$
A Proof with Quantifiers

If $x$ is even, then $x^2$ is even.

definition even (x : nat) := $\exists$ k, x = 2 * k

theorem x_even_x2_even (x: nat) (h: even x) : even (x * x) :=
  exists.elim h
  (assume k,
   assume hk : x = 2 * k,
   show even (x * x),
   from exists.intro (k * x)
   (calc x * x = (2 * k) * x : by rw hk
    ... = 2 * (k * x) : by rw nat.mul_assoc
   )
  )
Calculational Proofs and Tactics

calc \( x \cdot x = (2 \cdot k) \cdot x \): by rw hk
    ...
    = 2 \cdot (k \cdot x) : by rw nat.mul_assoc

Calc is a special proof mode for “calculation”
  • Proofs that involve the transitivity of equality

  • At each step we must show the justification for the equality
    • rw stands for “rewrite”, any rule that involves an algebraic rewrite
    • rw hk means a substitution using the type of hk (recall hk: x = 2 \cdot k)
    • rw nat.mul_assoc means apply the associativity law for multiplication \((x \cdot y)\cdot z = x \cdot (y \cdot z)\)

  • Lean automates some patterns of rules (tactics)
Summary

• There are many more features of Lean
  • Many other propositions, functions, and proof combinators
  • Lots of libraries
  • Many other alternative shorthands

• But this much is a good starting point
  • You will need to learn more from the documentation
  • The Lean tutorial is quite good!

• With practice, writing proofs becomes like programming
  • Dependent type theory shows, in fact, that it is just programming!