The Lean Proof Assistant
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)

• 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:

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

The `#check` command prints out information about a name
- Useful for debugging

```plaintext
#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

\[ \frac{A \vdash e_1 : t \to t'}{A \vdash e_1 e_2 : t'} \quad \text{[App]} \]

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

\[ \frac{A, x : t \vdash e : t'}{A \vdash \lambda x. e : t \to t'} \quad \text{[Abs]} \]

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

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

• Two equivalent definitions of a function:

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

• 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 λx, e not λx. e
  • A list of variables is separated by spaces, not commas
    • Parentheses 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

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:

```agda
consants p q : Prop
definition t1 : p -> q -> p := λ hp : p, λ hq : q, hp
```

- But Prop = Type
- And `def!` = theorem
- 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 `\lambda`
The Polymorphic Version

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

```
theorem t3 (p,q: Prop) : p → q → p :=
  assume hp : p,
  assume hq : q,
  hp
```
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 \( p \land q \rightarrow q \land p \)

\[
\text{lemma a5 (hpq: p \land q) : q \land p := and.intro (and.right hpq) (and.left hpq)}
\]

\[
\text{lemma a6 (hpq: p \land q) : q \land p := and.intro hpq.right hpq.left}
\]

\[
\text{lemma a7 (hpq: p \land q) : q \land p := \langle hpq.right, hpq.left \rangle}
\]
Disjunction: Or Introduction

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

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

lemma o2 : q \rightarrow p \lor 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 $r = q \lor p$
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 \((f_1: p -> q)\) \((f_2: p -> r)\) \((x:p)\) : q \land r :=

have hq: q, from f1 x,
have hr: r, from f2 x,
show q \land r, from \(\langle \text{hq}, \text{hr} \rangle\)
Quantifiers

• We’ve already seen examples of universal quantifiers

• Recall
  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

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

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

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

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

```
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, $x = 2 \times k$

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 * x = (2 * k) * x : by rw hk
    ...
    = 2 * (k * 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 * k)
    • rw nat.mul_assoc means apply the associativity law for multiplication (x * y)* z = x * (y * 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

- With practice, writing proofs becomes like programming
  - Dependent type theory shows, in fact, that it is just programming!
Final Thoughts
The Big Picture: Language Goals

Productivity:
- Python
- Matlab, NumPy
- Java, C++

Safety:
- Coq, Lean
- ML, Haskell

Performance:
- Rust
- C

Alex Aiken  CS 242  Lecture 1
Language Goals

• Every programming language has as goals
  • Performance
  • Productivity
  • Safety

• But there are tradeoffs

• And different designs make different choices
  • One of the reasons we have so many programming languages
Tradeoffs: Productivity vs. Safety
Proving Properties of Programs

<table>
<thead>
<tr>
<th>Automatic, Low complexity</th>
<th>Automatic, High complexity</th>
<th>Automatic or Semi-automatic Often undecidable</th>
<th>Manual, Undecidable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simply Typed Lambda Calculus</td>
<td>Static Analysis</td>
<td>Invariant Inference</td>
<td>Dependent Types</td>
</tr>
</tbody>
</table>

Alex Aiken  CS 242  Lecture 18
Tradeoffs: Productivity vs. Safety
Proving Properties of Programs

Automatic, Low complexity
Simply Typed
Lambda Calculus
Every typed language
Gradual Types
Every optimizing compiler

Automatic, High complexity
Static Analysis

Automatic or Semi-automatic
Often undecidable
Invariant Inference
Still figuring this part out ...

Manual, Undecidable
Dependent Types
Emerging from the lab ...
Tradeoffs: Productivity vs. Performance

• Array programming languages support both!

• But ...
  • Limited to arrays
  • First-order – no higher order functions, no objects ...
# Tradeoffs: Performance vs. Safety

10 Versions of Matrix Multiply from Leiserson & Shun

<table>
<thead>
<tr>
<th>Version</th>
<th>Implementation</th>
<th>Running time (s)</th>
<th>Relative speedup</th>
<th>Absolute Speedup</th>
<th>GFLOPS</th>
<th>Percent of peak</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Python</td>
<td>21041.67</td>
<td>1.00</td>
<td>1</td>
<td>0.006</td>
<td>0.001</td>
</tr>
<tr>
<td>2</td>
<td>Java</td>
<td>2387.32</td>
<td>8.81</td>
<td>9</td>
<td>0.058</td>
<td>0.007</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>1155.77</td>
<td>2.07</td>
<td>18</td>
<td>0.118</td>
<td>0.014</td>
</tr>
<tr>
<td>4</td>
<td>+ interchange loops</td>
<td>177.68</td>
<td>6.50</td>
<td>118</td>
<td>0.774</td>
<td>0.093</td>
</tr>
<tr>
<td>5</td>
<td>+ optimization flags</td>
<td>54.63</td>
<td>3.25</td>
<td>385</td>
<td>2.516</td>
<td>0.301</td>
</tr>
<tr>
<td>6</td>
<td>Parallel loops</td>
<td>3.04</td>
<td>17.97</td>
<td>6,921</td>
<td>45.211</td>
<td>5.408</td>
</tr>
<tr>
<td>7</td>
<td>+ tiling</td>
<td>1.79</td>
<td>1.70</td>
<td>11,772</td>
<td>76.782</td>
<td>9.184</td>
</tr>
<tr>
<td>8</td>
<td>Parallel divide–and–conquer</td>
<td>1.30</td>
<td>1.38</td>
<td>16,197</td>
<td>105.722</td>
<td>12.646</td>
</tr>
<tr>
<td>9</td>
<td>+ compiler vectorization</td>
<td>0.70</td>
<td>1.87</td>
<td>30,272</td>
<td>196.341</td>
<td>23.486</td>
</tr>
<tr>
<td>10</td>
<td>+ AVX intrinsics</td>
<td>0.39</td>
<td>1.76</td>
<td>53,292</td>
<td>352.408</td>
<td>41.677</td>
</tr>
</tbody>
</table>
# Tradeoffs: Performance vs. Safety

> #10 is much more complicated than #1!

<table>
<thead>
<tr>
<th>Version</th>
<th>Implementation</th>
<th>Running time (s)</th>
<th>Relative speedup</th>
<th>Absolute Speedup</th>
<th>GFLOPS</th>
<th>Percent of peak</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Python</td>
<td>21041.67</td>
<td>1.00</td>
<td>1</td>
<td>0.006</td>
<td>0.001</td>
</tr>
<tr>
<td>2</td>
<td>Java</td>
<td>2387.32</td>
<td>8.81</td>
<td>9</td>
<td>0.058</td>
<td>0.007</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>1155.77</td>
<td>2.07</td>
<td>18</td>
<td>0.118</td>
<td>0.014</td>
</tr>
<tr>
<td>4</td>
<td>+ interchange loops</td>
<td>177.68</td>
<td>6.50</td>
<td>118</td>
<td>0.774</td>
<td>0.093</td>
</tr>
<tr>
<td>5</td>
<td>+ optimization flags</td>
<td>54.63</td>
<td>3.25</td>
<td>385</td>
<td>2.516</td>
<td>0.301</td>
</tr>
<tr>
<td>6</td>
<td>Parallel loops</td>
<td>3.04</td>
<td>17.97</td>
<td>6,921</td>
<td>45.211</td>
<td>5.408</td>
</tr>
<tr>
<td>7</td>
<td>+ tiling</td>
<td>1.79</td>
<td>1.70</td>
<td>11,772</td>
<td>76.782</td>
<td>9.184</td>
</tr>
<tr>
<td>8</td>
<td>Parallel divide-and-conquer</td>
<td>1.30</td>
<td>1.38</td>
<td>16,197</td>
<td>105.722</td>
<td>12.646</td>
</tr>
<tr>
<td>9</td>
<td>+ compiler vectorization</td>
<td>0.70</td>
<td>1.87</td>
<td>30,272</td>
<td>196.341</td>
<td>23.486</td>
</tr>
<tr>
<td>10</td>
<td>+ AVX intrinsics</td>
<td>0.39</td>
<td>1.76</td>
<td>53,292</td>
<td>352.408</td>
<td>41.677</td>
</tr>
</tbody>
</table>
These tradeoffs explain why there are so many different languages
- But there are many fewer language building blocks
- Put together in endless variations

New language technology is always coming
- New ideas in programming
- Changes in underlying hardware
- Changes in needs (e.g., security)

We have focused on
- The building blocks of programming languages that have stood the test of time
- New and emerging ideas in programming
Thanks!