The Lean Proof Assistant

CS242

Lecture 18

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:

```
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

$$\frac{A \vdash e_1 : t \rightarrow t' \qquad A \vdash e_2 : t}{A \vdash e_1\, e_2 : t'} \quad \text{[App]}$$

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

From a proof of $t \rightarrow t'$ and 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:

```
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 := 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
    - Parens 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:

```
constants p q : Prop
theorem t1 : p -> q -> p := λ hp: p, λ hq : q, hp
```

- But Prop = Type
- And theorem = def!
- Just alternative syntax to emphasize proofs instead of computation

And More Options

- We could also write this proof

```
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

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

Conjunction: And Introduction

A few proofs of $p \to q \to p \wedge q$

```
lemma a1 (hp : p) (hq : q) : p ∧ q := and.intro hp hq
```

or

```
lemma a2 : p → q → p ∧ q := λ hp: p, λ hq : q, and.intro hp hq
```

or

```
lemma a3 : p → q → p ∧ q :=
    assume hp: p,
    assume hq: q,
    and.intro hp hq
```

or

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

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

Conjunction: And Elimination

Proofs of $p \wedge q \rightarrow q \wedge p$

```
lemma a5 (hpq: p ∧ q) : q ∧ p := and.intro (and.right hpq) (and.left hpq)

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

lemma a7 (hpq: p ∧ q) : q ∧ p := ⟨ hpq.right, hpq.left ⟩
```

Disjunction: Or Introduction

Proofs of p → p ∨ q and q → p ∨ 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 ∨ q → q ∨ p

```
lemma o3 (h : p ∨ q) : q ∨ 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 ∨ q

a function of type p → r

a function of type q → r

In this example r = q ∨ 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

$(p \to q) \to (p \to r) \to (p \to q \land r)$

```
lemma imp (f1: p -> q) (f2: p -> r) (x:p) : q ∧ r :=
  have hq: q, from f1 x,
  have hr: r, from f2 x,
  show q ∧ r, from ⟨ hq, hr ⟩
```

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: ∃ 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) = ∃ 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) := ∃ 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 * 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.mulassoc means apply the associativity law for multiplication (x * y)* z = x * (y * z)

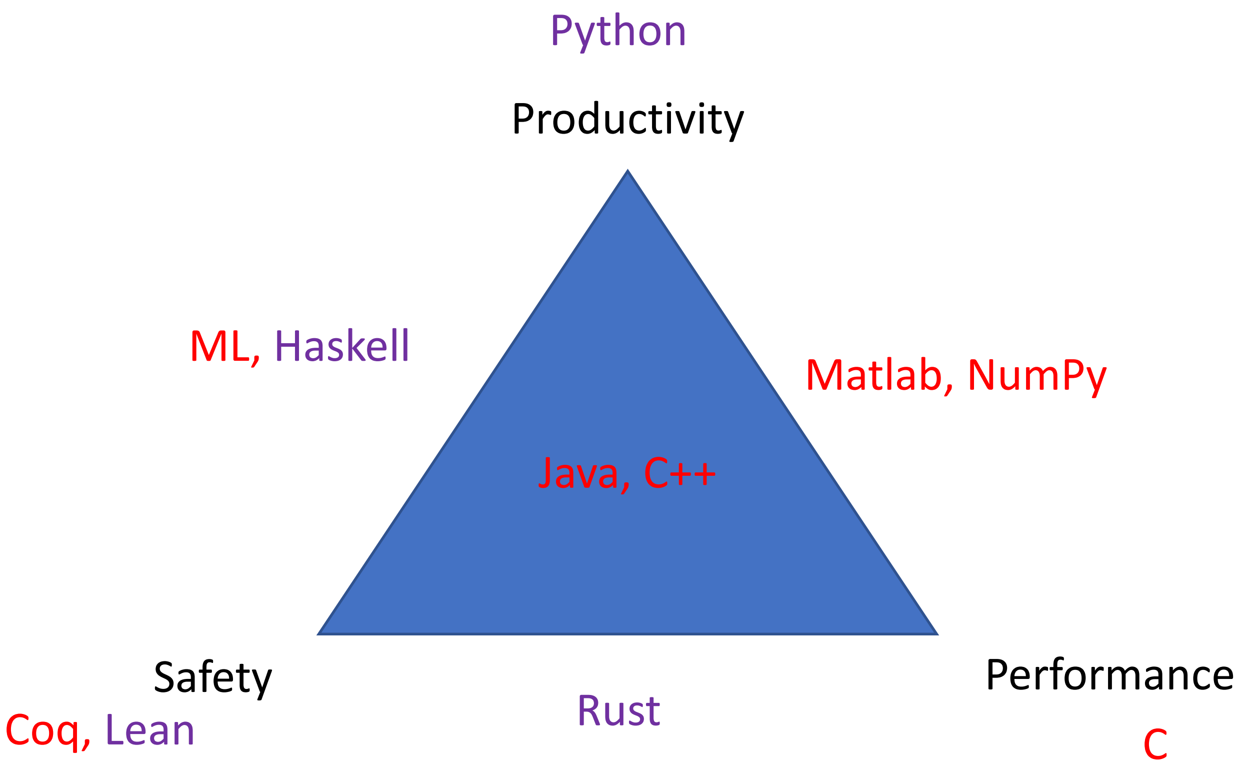
- Lean automates some patterns of rules (tactics)

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



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

Automatic, Low complexity	Automatic, High complexity	Automatic or Semi-automatic Often undecidable	Manual, Undecidable
Simply Typed Lambda Calculus	Static Analysis	Invariant Inference	Dependent Types

Tradeoffs: Productivity vs. Safety
Proving Properties of Programs

Automatic,
Low complexity

Automatic,
High complexity

Automatic or Semi-automatic
Often undecidable

Manual,
Undecidable

Simply Typed
Lambda Calculus

Every typed
language

Gradual Types

Static Analysis

Every optimizing
compiler

Invariant Inference

Still figuring this
part out …

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

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

Tradeoffs: Performance vs. Safety

#10 is much more complicated than #1 !

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

The Last Slide …

- 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!