Inductive Types

Static Analysis section, **Lecture 25**

Pavel Panchekha

CS 6110, U of Utah

9 April 2020

Proof rules

Introduction and elimination rules for **and**:

$$\frac{a \qquad b}{a \wedge b}$$

Introduction

$$\frac{a \wedge b}{a} \qquad \frac{a \wedge b}{b}$$

Elimination

These are **basically opposites**!

No “free information” from introducing intermediate steps

Implication

$$\frac{\Gamma, x : a \vdash e : b}{\Gamma \vdash (\lambda x : a, e) : a \to b}$$

$$\frac{\Gamma \vdash f : a \to b \qquad \Gamma \vdash e : a}{\Gamma \vdash (f\, e) : b}$$

Mixing the Two

If proofs are programs and propositions are types…

1. One language for proofs **and** programs
2. Syntax **reused** for programs and proofs
3. Data structures **mix** programs and proofs
4. Proof or program? **You decide!**

Class Progress

Logical reasoning

Program logics

Static analysis

Abstract Interpretation

Dependent Types

Propositions as Types

Inductive Types

Type Dependency

Inductive Types

Defining new **inductive** data types

Natural numbers, lists, and trees

Compiling recursion to induction

Finding recursive calls and replacing them

Proof by recursion over a data type

Constructing proofs recursively

Inductive Types

Natural numbers, lists, and trees

New Data Types

Pairs and functions have proof analogs

But what about natural numbers, lists, trees, etc?

Recall **axioms of Peano arithmetic**:

- Natural numbers are **0 or (n + 1)**
- **Define** plus / times via (n+1)
- **Induction axiom**

Constructors

Defines natural numbers via their **constructors**:

- **All natural numbers** are 0 or (n + 1)

$$\forall n, n = 0 \vee \exists n', n = n' + 1$$

- 0 and (n + 1) are **different**

$$\forall n', 0 \neq n' + 1$$

The Pattern

Defining “or” terms via their **constructors**:

- All “or” terms are **lft or rgt**

$$\forall c, (\exists a, c = \textbf{lft}\ a) \vee (\exists b, c = \textbf{rgt}\ b)$$

- **lft and rgt** are different

$$\forall a, \forall b, \textbf{lft}\ a \neq \textbf{rgt}\ b$$

The Pattern

Defining “and” terms via their **constructors**:

- All “and” terms are **(a, b)**

$$\forall c, \exists a, \exists b, c = (a, b)$$

- ~~**(a, b)** are different~~

~~???~~

Introduction Rules

Constructors **define introduction rules** for a data type:

$$\frac{}{0 : \textbf{nat}} \qquad \frac{n : \textbf{nat}}{n + 1 : \textbf{nat}}$$

What about **elimination rules**?

Analogy from OR

Or has **two constructors**:

nat has two constructors

$$\frac{p_a : a}{\textbf{lft}\ p_a : a \vee b} \qquad \frac{p_b : b}{\textbf{rgt}\ p_b : a \vee b}$$

The elimination rules has **two extra assumptions**:

$$\frac{f_a : a \to c \qquad p : a \vee b \qquad f_b : b \to c}{\textbf{cases}(p, f_a, f_b) : c}$$

Analogy for Nat

nat has two constructors:

$$\frac{}{0 : \textbf{nat}} \qquad \frac{n : \textbf{nat}}{n + 1 : \textbf{nat}}$$

The elimination rules has **two extra assumptions**:

$$\frac{f_0 : c \qquad n : \textbf{nat} \qquad f_{+1} : \textbf{nat} \to \textbf{nat}}{\textbf{rec}(n, f_0, f_{+1}) : c}$$

To Induction

$$\frac{f_0 : c \qquad n : \textbf{nat} \qquad f_{+1} : \textbf{nat} \to \textbf{nat}}{\textbf{rec}(n, f_0, f_{+1}) : c}$$

Meanwhile, the **induction rule**:

$$\frac{P(0) \qquad n : \textbf{nat} \qquad P(n) \to P(n+1)}{P(n)}$$

Key: replace **nat** by **a property of nat**

Compiling Recursion

Recursion via induction

Generalizing

Define new types **by defining their constructors**:

Constructors have **arguments**, including from that type

Each constructor is an **introduction rule**

Per-type **elimination rule** based on constructors

Function takes constructor arguments, returns c

Induction rule generalizes elimination rule

Replace output type c with property $P(\text{constructor})$

Recursive Definitions

Normal recursive function **examines constructors**:

```
def length : ∀τ, list τ → nat
| nil := 0
| (cons a r) := length r + 1
```

Define recursive function via **induction**:

$$\frac{P([]) \qquad l : \textbf{list}\ \tau \qquad \forall x, \forall l', P(l') \rightarrow P(x :: l')}{P(l)}$$

Recursive Definitions

Normal recursive function **examines constructors**:

```
def length : ∀τ, list τ → nat :=
λ l, list.rec l (λ l, nat)
(λ, 0)
(λ x l length_l, length_l + 1)
```

Define recursive function via **induction**:

$$\frac{P([]) \qquad l : \textbf{list}\ \tau \qquad \forall x, \forall l', P(l') \rightarrow P(x :: l')}{P(l)}$$

Uses of Induction

Induction can **prove properties of all numbers**

- Prove base case, $P(0)$
- Prove inductive case, $P(n) \rightarrow P(n+1)$

Induction can also **compute recursive function**

- Implement base case, $f(0)$
- Implement recursive case, $f(n+1)$, in terms of $f(n)$

Uses of Induction

Induction can **prove properties of all numbers**

- Prove base case, $P(0)$
- Prove inductive case, $P(n) \to P(n+1)$

Induction can also **compute recursive function**

- Implement base case, $f(0)$
- Implement recursive case, $f(n) \mapsto f(n+1)$

Course Updates

Final Projects

Final Presentation

Final project presentations **due Thursday**

- **Double the normal length** (8 minutes solo, 16 minutes group)
- **Record and upload** the presentations
- Watch and **submit questions** for others' presentations

Presentations should **stand alone**

- Cover **every part** of the project; show what you **achieved**
- Discuss how you **checked that the results are good**

Final Project

Submit code for final project same day as presentations

- We'll **run quicksort** on test cases
- We'll **verify programs**, and try to verify false things

Group projects will be graded by **reading code**

If you're behind due to coronavirus, email me

Proof by Recursion

Proofs as Programs

A simple proof

Consider proving this theorem:

$\forall n, 0 \leq n$

```
def t : ∀n, 0 <= n
```

Let's prove it **by induction**:

```
nat.rec_on n
```

$P(0) = 0 \leq 0$

```
le.refl 0 : 0 <= 0
```

$P(n) \to P(n+1) = 0 \leq n \to 0 \leq (n+1)$

```
λ n pf, le.step pf
```

A simple proof

Proof by induction, as code:

```
theorem t : ∀ n, 0 ≤ n :=
λ n, nat.rec_on n
  (le.refl 0)
  (λ n' pf, le.step pf)
```

“Uncompile” to **recursive function**, as code:

```
def t : ∀ n, 0 ≤ n
| 0 := le.refl 0
| (n + 1) := le.step (t n)
```

Recursion + Induction

Proof by induction is **recursively computing a proof**

Prove $P(0)$ → Implement $f(0)$ (of type $P(0)$)

Prove $P(n) \to P(n+1)$ → Implement $f(n) \mapsto f(n+1)$

When proofs are programs, **induction is recursion**!

Next class:

Type Dependency

To do:

- ☐ Course feedback
- ☐ Class projects

Inductive Types

Defining new **inductive** data types

Natural numbers, lists, and trees

Compiling recursion to induction

Finding recursive calls and replacing them

Proof by recursion over a data type

Constructing proofs recursively

PREDICATES

PROPOSITIONS

ARE

TYPES TOO

INDUCTION

ON

PROOFS

DEPENDENCY

COORDINATING

MULTIPLE

TERMS

Next class:

Type Dependency

To do:

- ☐ Course feedback
- ☐ Class projects