Induction and Recursion

CAS CS 491: Type Theory and Mechanized Reasoning

January 24, 2023

Outline

Administrivia

Assignment 1 will be released tomorrow. It will be very short, just a couple problems about induction, maybe a bit of OCaml programming.

I will create a Slack channel later this week.

We still haven't gotten a course number change. And I'm still not 100% sure if it will be run. I'm pressing the department a bit more to give me more information.

Goals and Motivations for Today

Make sure we have the mathematical background necessary for this coures. Induction is probably the most important proof techniques when it comes to using proof assistants like Lean.

Think a bit about what induction actually is, and how we should think about it in a formalized setting.

If we have some time, look into more interesting inductively defined collections, in particular generalized algebraic data types in OCaml.

Preamble: What do we take for granted?

When we *do* mathematics, there's a lot more that we take for granted than we might expect. For example, in complexity theory and algorithm design, when we prove that an algorithm is polynomial time, *what does this mean*?

One of the meta-objectives of formalizing reasoning is thinking about what we take for granted. When a computer is checking our proofs, we're *not allowed* to take anything for granted.

- In the cases where we do a lot of "hand-wavy" reasoning how do we make it formal?
- In the case that its (seemingly) impossible, can we re-evaluate the reasoning we're doing to make it more amenable to formalization?
- What things should be formalized?

Preamble: What do we take for granted?

When we *do* mathematics, there's a lot more that we take for granted than we might expect. For example, in complexity theory and algorithm design, when we prove that an algorithm is polynomial time, *what does this mean*?

One of the meta-objectives of formalizing reasoning is thinking about what we take for granted. When a computer is checking our proofs, we're *not allowed* to take anything for granted.

- In the cases where we do a lot of "hand-wavy" reasoning how do we make it formal?
- In the case that its (seemingly) impossible, can we re-evaluate the reasoning we're doing to make it more amenable to formalization?
- What things should be formalized?

Preamble: What do we take for granted?

When we *do* mathematics, there's a lot more that we take for granted than we might expect. For example, in complexity theory and algorithm design, when we prove that an algorithm is polynomial time, *what does this mean*?

One of the meta-objectives of formalizing reasoning is thinking about what we take for granted. When a computer is checking our proofs, we're *not allowed* to take anything for granted.

- In the cases where we do a lot of "hand-wavy" reasoning how do we make it formal?
- In the case that its (seemingly) impossible, can we re-evaluate the reasoning we're doing to make it more amenable to formalization?
- What things should be formalized?

Outline

What is induction?

Informal Definition. Induction is a mathematical principle for showing that a property holds of every thing in an inductively defined collection.

- *What is a mathematical principle?*
- *Why are we allowed to used induction?*

An Aside: Peano Arithmetic

Peano Arithmetic is a theory for working with natural numbers in a formal way.

Induction is an axiom in theory of Peano Arithmetic. This mean we have to assume that we can do induction.

Questions:

- *Why do we get to assume that?*
- *What if we don't assume it?*

An Aside: Peano Arithmetic

Peano Arithmetic is a theory for working with natural numbers in a formal way.

Induction is an axiom in theory of Peano Arithmetic. This mean we have to assume that we can do induction.

Questions:

- *Why do we get to assume that?*
- *What if we don't assume it?*

An Aside: Peano Arithmetic

Peano Arithmetic is a theory for working with natural numbers in a formal way.

Induction is an axiom in theory of Peano Arithmetic. This mean we have to assume that we can do induction.

Questions:

- *Why do we get to assume that?*
- *What if we don't assume it?*

Induction over Natural Numbers

The Induction Principle. In order to show that a property P holds of every natural number, it suffices to show that

- P holds of 0
- for any natural number i, if P holds of i, then it also holds of $i + 1$.

Questions:

- *What is a property?*
- *How can we tell that a property holds?*

Induction over Natural Numbers

The Induction Principle. In order to show that a property P holds of every natural number, it suffices to show that

- P holds of 0
- for any natural number i, if P holds of i, then it also holds of $i + 1$.

Questions:

- *What is a property?*
- *How can we tell that a property holds?*

An Aside: Induction is Second Order

When we state the induction principle, we're really saying something a bit stronger:

The Induction Principle. For any property P, in order to show that a property P holds of every natural number, it suffices to show that

- P holds of 0
- for any natural number i, if P holds of i, then it also holds of $i + 1$.

Questions:

- *How do we formalize the idea of every property?* In Peano arithmetic, there are actually infinitely many induction principles one for each property P.

An Aside: Induction is Second Order

When we state the induction principle, we're really saying something a bit stronger:

The Induction Principle. For any property P, in order to show that a property P holds of every natural number, it suffices to show that

- P holds of 0
- for any natural number i, if P holds of i, then it also holds of $i + 1$.

Questions:

- *How do we formalize the idea of every property?* In Peano arithmetic, there are actually infinitely many induction principles one for each property P.

An Aside: Induction is Second Order

When we state the induction principle, we're really saying something a bit stronger:

The Induction Principle. For any property P, in order to show that a property P holds of every natural number, it suffices to show that

- P holds of 0
- for any natural number i, if P holds of i, then it also holds of $i + 1$.

Questions:

- *How do we formalize the idea of every property?* In Peano arithmetic, there are actually infinitely many induction principles one for each property P.

Warm-Up Example

For every natural number n,

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Domino Analogy

Consider the following thought experiment: we set up an infinitely long line of dominoes along the number line, one on each natural number 0, 1, 2, and so on. Then we tip the domino at 0.

We can prove with the induction principle that every domino will eventually fall.

More Challenging Example

Bernoulli's Inequality. For every natural number n, if $h > -1$ then $1 + nh \leq (1 + h)^n$.

Strong Induction Over Natural Numbers

The restriction that we prove that the property holds of $i + 1$ *only* from i is unnecessarily restrictive.

In the domino analogy, if certainly know that *all* the dominoes before domino i fall, then domino i falls as well.

Strong Induction Principle. To prove that a property P holds of all natural numbers, it suffices to show:

- for any natural number i, if P holds of *all* j where $j < i$, then P holds of i.

Strong Induction Over Natural Numbers

The restriction that we prove that the property holds of $i + 1$ *only* from i is unnecessarily restrictive.

In the domino analogy, if certainly know that *all* the dominoes before domino i fall, then domino i falls as well.

Strong Induction Principle. To prove that a property P holds of all natural numbers, it suffices to show:

- for any natural number i, if P holds of *all* j where $j < i$, then P holds of i.

Strong Induction Over Natural Numbers

The restriction that we prove that the property holds of $i + 1$ *only* from i is unnecessarily restrictive.

In the domino analogy, if certainly know that *all* the dominoes before domino i fall, then domino i falls as well.

Strong Induction Principle. To prove that a property P holds of all natural numbers, it suffices to show:

- for any natural number i, if P holds of *all* j where $j < i$, then P holds of i.

Warm-Up Example

Every natural number n greater than 2 can be expressed as a product of primes.

Strong Induction vs. Ordinary Induction

At this point, it is natural to wonder:

Why use induction over strong induction?

In mathematical settings, this is a fair question. Strong induction implies ordinary induction.

In formalized reasoning, it can make a big difference. . .

What do we assume, without thinking about it, in our proof of the previous example?

Strong Induction vs. Ordinary Induction

At this point, it is natural to wonder:

Why use induction over strong induction?

In mathematical settings, this is a fair question. Strong induction implies ordinary induction.

In formalized reasoning, it can make a big difference. . .

What do we assume, without thinking about it, in our proof of the previous example?

Strong Induction vs. Ordinary Induction

At this point, it is natural to wonder:

Why use induction over strong induction?

In mathematical settings, this is a fair question. Strong induction implies ordinary induction.

In formalized reasoning, it can make a big difference. . .

What do we assume, without thinking about it, in our proof of the previous example?

The Point

Ordinary Induction is structural and easier to reason about formally.

More Challenging Example

$\sqrt{2}$ is irrational.

Outline

Inductively Defined Sets

If this we're a different course, we might spend of a lot of time defining inductive sets and recursion.

You may have seen Backus-Naur Form (BNF) grammars (in a course like CS320) as a way of defining inductive collections, e.g.,

$$n \in \mathbb{N} ::= Z \mid Sn$$

In this course, we'll take a very simple definition: *an inductive collection is defined by the terms of an algebraic data type.*

Inductively Defined Sets

If this we're a different course, we might spend of a lot of time defining inductive sets and recursion.

You may have seen Backus-Naur Form (BNF) grammars (in a course like CS320) as a way of defining inductive collections, e.g.,

$$n \in \mathbb{N} ::= Z \mid Sn$$

In this course, we'll take a very simple definition: *an inductive collection is defined by the terms of an algebraic data type.*

Inductively Defined Sets

If this we're a different course, we might spend of a lot of time defining inductive sets and recursion.

You may have seen Backus-Naur Form (BNF) grammars (in a course like CS320) as a way of defining inductive collections, e.g.,

$$n \in \mathbb{N} ::= Z \mid Sn$$

In this course, we'll take a very simple definition: *an inductive collection is defined by the terms of an algebraic data type.*

Algebraic Data Types in OCaml

The natural numbers:

```
type nat
= Z
| S of nat
```

Values of type nat:

```
let zero  = Z
let one   = S Z
let two   = S (S Z)
let three = S (S (S Z))
let four  = S three
```

A natural number is an *abstract* object which *represents* the number of "S" constructors in the value.

Algebraic Data Types in OCaml

The natural numbers:

```
type nat
= Z
| S of nat
```

Values of type `nat`:

```
let zero  = Z
let one   = S Z
let two   = S (S Z)
let three = S (S (S Z))
let four  = S three
```

A natural number is an *abstract* object which *represents* the number of "S" constructors in the value.

Algebraic Data Types in OCaml

The natural numbers:

```
type nat
= Z
| S of nat
```

Values of type `nat`:

```
let zero  = Z
let one   = S Z
let two   = S (S Z)
let three = S (S (S Z))
let four  = S three
```

A natural number is an *abstract* object which *represents* the number of "S" constructors in the value.

Algebraic Data Types in OCaml (Continued)

Lists:

```
type 'a list
= Nil (* like Z *)
| Cons of 'a * 'a list (* like S *)
```

Values of type `int list`:

```
let nil : int list = Nil
let l : int list = Cons (1, Cons(2, Nil))
```

A list is like a natural number in which each "`S`" also holds a piece of data.

Recursion over Algebraic Data Types

Now it is very easy to define what recursion is over inductive sets: it's just pattern matching.

Examples:

```
let rec int_of_nat n =
  match n with
  | Z -> 0
  | S k -> 1 + int_of_nat k

let rec length l =
  match l with
  | Nil -> 0
  | Cons (x, xs) -> 1 + length xs
```

The Point

Inductively-define collections are algebraic data types in this course.

Recursion over inductively-defined collections is pattern matching.

Outline

What is structural induction?

Structural induction is the generalization of induction to inductively defined sets (i.e., algebraic data types).

Principle of Structural Induction. In order to prove that a property P holds of all values of an ADT, it suffices to prove

- P holds for all *ground* constructors
- P holds for all *recursive* constructors Tag, assuming it holds of all constructor that Tag depends on

What is structural induction?

Structural induction is the generalization of induction to inductively defined sets (i.e., algebraic data types).

Principle of Structural Induction. In order to prove that a property P holds of all values of an ADT, it suffices to prove

- P holds for all *ground* constructors
- P holds for all *recursive* constructors `Tag`, assuming it holds of all constructor that `Tag` depends on

Warm-Up Example

```
type nat
= Z
| S of nat

let add m n =
  match m with
  | Z -> n
  | S k -> S (add k n)
```

Show that `add n 0` is `n` for all natural numbers `n`.

Structural induction and verification

If you write a lot of functional code you're likely going to use a lot of ADTs. This makes structural induction *incredibly important* from the perspective of verifying the correctness of code.

In other words: Structural induction is used to *prove things about* inductively-defined collections and functions on them.

(And we'll often find that the proof will follow the structure of the functions we prove things about)

More Challenging Example

```
type 'a tree
= Empty
| Node of 'a tree * 'a tree

let depth t =
  match t with
  | Empty -> 0
  | Node (l, r) -> 1 + Int.max (depth l) (depth r)

let size t =
  match t with
  | Empty -> 0
  | Node (l, r) -> 1 + l + r
```

Show that `depth t` is at most `size t`.

Final Remarks on Induction

You could write a thesis on induction... Here are a couple other things you might think about:

- What if you could only do induction on *some* properties, but not all?
- What if you could do induction for *larger* infinite inductively-defined sets?

Looking forward a bit: We'll start looking at Lean next week. We will come to realize that, in lean, induction *is* recursion.

Final Remarks on Induction

You could write a thesis on induction. . . Here are a couple other things you might think about:

- What if you could only do induction on *some* properties, but not all?
- What if you could do induction for *larger* infinite inductively-defined sets?

Looking forward a bit: We'll start looking at Lean next week. We will come to realize that, in lean, induction *is* recursion.

Outline

Motivation: Lean as a Functional Programming Language

Next week we'll start looking at Lean as a functional programming language with "strange features".

Eventually we'll see these features allow for theorem proving.

Generalized Inductive Definitions

Let's maybe look at some examples.

Outline

Summary

Induction is a mathematical principle for proving that a property holds of every element of an inductively-defined set.

It is used to prove things about algebraic data types.

Moving forward, we will have to think carefully what assumptions we make when we prove things on paper.