Inductive data

Robert Y. Lewis

CS 0220 2024

April 24, 2024
Overview

1. Why does induction work?
2. Other inductive sets
3. Inductive proofs
4. Closing
Induction on $\mathbb{N}$

We introduced induction as a technique to prove things about natural numbers.

It makes some intuitive sense. But let’s examine things more carefully.
Defining \( \mathbb{N} \)

What are the natural numbers?

1. 0 is a natural number.
2. For any natural number \( k \), \( k + 1 \) is a natural number. \( \text{successor}(k) \)
3. Successor is injective.
4. For every \( k \), \( \text{successor}(k) \neq 0 \).
5. Every natural number is either 0 or the successor of another natural number.

Are there any sets that satisfy properties 1-5 that don’t look like \( \mathbb{N} \)?
Defining \(\mathbb{N}\)

Let’s try again.

1. 0 is a natural number.
2. For any natural number \(k\), \(k + 1\) is a natural number. \(\text{successor}(k)\)
3. \(\text{successor}\) is injective.
4. For every \(k\), \(\text{successor}(k) \neq 0\).
5. Every natural number can be represented as a (finite) directed tree, where each node is either
   - labeled 0, and has no children; or
   - labeled successor, and has one child.

Condition 5 is equivalent to the principle of induction.
Again, succinctly

We define $\mathbb{N}$ to be an *inductive set* with *constructors*

- $0 : \mathbb{N}$
- successor: $\mathbb{N} \rightarrow \mathbb{N}$.

An inductive set is defined by giving a list of constructors that are assumed to satisfy properties 3-5. See also: *inductive types* or *algebraic data types* in some programming languages.

Inductive sets are sets of “discrete objects.”
And, recursion

Let $A$ be any set, $a \in A$, and $g : \mathbb{N} \times A \rightarrow A$. There exists a unique function $f : \mathbb{N} \rightarrow A$ satisfying the two clauses:

- $f(0) = a$
- $f(k + 1) = g(k, f(k))$

“Exists” and “unique.” In other words: “to define a function with domain $\mathbb{N}$, we can describe its behavior on the two constructors.”

Sounds a lot like induction. And the tree property.
Inductive lists

Let $A$ be a set. The set $L(A)$ of lists of elements of $A$ is an inductive set with constructors

- $\text{nil} : L(A)$
- $\text{cons} : A \times L(A) \rightarrow L(A)$

“To create a list, either create the empty list, or take a list and tack on one more value.”
Induction on lists

- nil : $L(A)$
- cons : $A \times L(A) \rightarrow L(A)$

Tree property?

Induction principle?

To show that $P(l)$ holds for every list $l \in L(A)$, show:
- $P(nil)$
- For every $a \in A$ and $l \in L(A)$, $P(l) \rightarrow P(cons(a, l))$
Inductive integers?

Let’s try to define \( \mathbb{Z} \) as an inductive set.

Constructors:

- \( 0 : \mathbb{Z} \)
- \( \text{successor} : \mathbb{Z} \rightarrow \mathbb{Z} \)
- \( \text{predecessor} : \mathbb{Z} \rightarrow \mathbb{Z} \)

Fails: why?
Inductive integers!

A working, if lame, attempt:

Constructors:

- $0 : \mathbb{Z}$
- $\text{posOfNat} : \mathbb{N} \rightarrow \mathbb{Z}$
- $\text{negOfNat} : \mathbb{N} \rightarrow \mathbb{Z}$

$\text{posOfNat}(n) \;"=\; n + 1$
$\text{negOfNat}(n) \;"=\; -(n + 1)$
Inductive formulas

The set $F$ of formulas of propositional logic is an inductive set with constructors

- $\text{letter} : \mathbb{N} \rightarrow F$
- $\text{not} : F \rightarrow F$
- $\text{and} : F \times F \rightarrow F$
- $\text{or} : F \times F \rightarrow F$
- $\text{implies} : F \times F \rightarrow F$
- $\text{iff} : F \times F \rightarrow F$

Principle of induction? To prove $P(\varphi)$ holds for every prop formula $\varphi$, it suffices to show:

- $P(\text{letter}(i))$ for every $i$ (“$P$ holds of every propositional letter”)
- $P(\varphi) \rightarrow P(\text{not}(\varphi))$
- $P(\varphi_1) \land P(\varphi_2) \rightarrow P(\text{and}(\varphi_1, \varphi_2))$
- $P(\varphi_1) \land P(\varphi_2) \rightarrow P(\text{or}(\varphi_1, \varphi_2))$
- $\ldots$
Inductive formulas

Recursion on formulas, in words:

To define a function $f : F \to A$, it suffices to describe the behavior of $F$ on each constructor of $F$.

Example: evaluation $E(\varphi)$ under a propositional assignment $\nu : \mathbb{N} \to \{T, F\}$.

- $E(\text{letter}(i)) = \nu(i)$
- $E(\text{not}(\varphi)) = \text{NOT}(E(\varphi))$
- $E(\text{and}(\varphi_1, \varphi_2)) = \text{AND}(E(\varphi_1), E(\varphi_2))$

Challenge: phrase this like we phrased recursion on $\mathbb{N}$. 
Proofs as data

We have a technique for figuring out if a propositional formula is valid: write the truth table, see if all columns are T.

This is more of a “process” than an “object.” Intuition: if you write down an argument like this, the only way I can check it is by doing it myself and comparing.

Other ways?
Proofs as data: introduction rules

How can I prove \( \varphi_1 \land \varphi_2 \)? Prove \( \varphi_1 \) and then prove \( \varphi_2 \).

How can I prove \( \varphi_1 \lor \varphi_2 \)? Prove \( \varphi_1 \). Alternatively, prove \( \varphi_2 \).

This sounds sort of inductive. Constructors?

- \textit{and\_intro} : \( \text{proof}(\varphi_1) \times \text{proof}(\varphi_2) \rightarrow \text{proof}(\varphi_1 \land \varphi_2) \)
- \textit{or\_intro\_left} : \( \text{proof}(\varphi_1) \rightarrow \text{proof}(\varphi_1 \lor \varphi_2) \)
- \textit{or\_intro\_right} : \( \text{proof}(\varphi_2) \rightarrow \text{proof}(\varphi_1 \lor \varphi_2) \)
Proofs as data: elimination rules

What can I do with a proof of $\varphi_1 \land \varphi_2$? Prove $\varphi_1$. Alternatively, prove $\varphi_2$.

- **and_elim_left**: $\text{proof}(\varphi_1 \land \varphi_2) \rightarrow \text{proof}(\varphi_1)$
- **and_elim_right**: $\text{proof}(\varphi_1 \land \varphi_2) \rightarrow \text{proof}(\varphi_2)$

What can I do with a proof of $\varphi_1 \lor \varphi_2$? Case split…tricky.

Need to analyze implication first, which also muddies the picture a bit.
Can’t dive into the details now. But we can make things more or less work.

*Induction on proofs?* “I can only construct proofs of valid formulas.”

*Recursion on proofs?* Given a proof, reconstruct the formula it proves—proof checking!

Proofs are directed trees!
Final thoughts

We’ve seen a lot of topics this semester. Remember why we’ve done this:

- Vocabulary. Use the languages of logic, combinatorics, probability, … as a shared, precise vocabulary for discussing problems.
- Abstraction. A lot of the problems we’ve studied will show up in different contexts, in and out of computer science. Remember our abstract solutions and adapt them to reality.
- Team problem solving. CS is collaborative, and hopefully you’ve gotten practice solving problems with a team.