Induction

Simon Oddershede Gregersen
gregersen@cs.au.dk

Based on lecture notes by Andrew C. Myers

September, 2018

Induction is a powerful tool for reasoning in mathematics. In the study of programming languages, it is frequent to encounter inductive reasoning in the form of inductive definitions and inductive proofs. In this note, we present two inductive reasoning techniques, namely *mathematical induction* and *structural induction*.

1 Mathematical Induction

Induction is a method of proving statements about inductively defined sets. A set is inductively defined when it is generated from some *base elements* using some set of *constructor operations*.

The most common example of an inductively defined set is the set of natural numbers

$$\mathbb{N} = \{0, 1, 2, \ldots\}.$$

The set can be generated from the base element 0 and the successor function *succ* where $succ(x) = x + 1$. That is, the natural numbers can be inductively defined by:

1. $0 \in \mathbb{N}$,
2. If $n \in \mathbb{N}$ then $n + 1 \in \mathbb{N}$.

Nothing else is an element of $\mathbb{N}$. Induction over the natural numbers is often called mathematical induction. There are many inductively defined sets other than the natural numbers, such as lists, trees and MiniScala expressions. Section 2 considers induction over such sets.

Example 1. Recall the concept of functional abstraction. After getting tired of expressions like $2 \cdot 2$, $42 \cdot 42$, and $123456789 \cdot 123456789$, we can package up the common part of these expressions using functional abstraction:

```
def square(x: Int): Int = x * x
```

In a sense, `square` packages up infinitely many potential computations $x \cdot x$ in a single finite box.

Now, we can do the same sort of abstraction with logical arguments. Induction is an example of logical abstraction. It essentially allows us to do infinitely many concrete arguments in a single abstract argument. For example, suppose we had no built-in integer multiplication and had to construct it using addition. Consider the following recursive program for multiplication of nonnegative integers:

```
def mult(m: Int, n: Int): Int =
  if (n == 0) 0 else m + mult(m, n - 1)
```

In arguing that this program is correct, we might go through the following thought process for `mult(m, n)`:

- For any integer `m`, the program works if `n` $= 0$:

$$\begin{aligned} \texttt{mult(m, 0)} &= \texttt{if (0 == 0) 0 else m + mult(m, 0 - 1)} \\ &= 0. \end{aligned}$$

- If `n` $= 1$, the program calls itself recursively on input 0. We just argued that it works for 0, and if we supply the correct answer in the recursive call, we can argue that it works for `n` $= 1$:

$$\begin{aligned} \texttt{mult(m, 1)} &= \texttt{if (1 == 0) 0 else m + mult(m, 1 - 1)} \\ &= \texttt{m + mult(m, 0)} \\ &= \texttt{m + 0} \\ &= \texttt{m}. \end{aligned}$$

- If `n` $= 2$, the program calls itself recursively on input 1. We just argued that it works for 1, and if we supply the correct answer in the recursive call, we can argue that it works for `n` $= 2$:

$$\begin{aligned} \texttt{mult(m, 2)} &= \texttt{if (2 == 0) 0 else m + mult(m, 2 - 1)} \\ &= \texttt{m + mult(m, 1)} \\ &= \texttt{m + m} \\ &= \texttt{2m}. \end{aligned}$$

- If `n` $= 3$,the programs calls itself recursively on input 2. We just argued that it works for 2, and if we supply the correct answer in the recursive call, we can argue that it works for `n` $= 3$:

$$\begin{aligned} \texttt{mult(m, 3)} &= \texttt{if (3 == 0) 0 else m + mult(m, 3 - 1)} \\ &= \texttt{m + mult(m, 2)} \\ &= \texttt{m + 2m} \\ &= \texttt{3m} \end{aligned}$$

and so on. Note that the argument for the base case 0 is different but after that, the arguments all look the same. It is only the numbers that differ. Using logical abstraction, we can do all these infinitely many cases at once. For any `n` > 0, the program calls itself recursively on input `n - 1`. Assume that we have shown that it works for `n - 1`. If we supply the correct answer in the recursive call, we can argue that it works for `n`:

$$\begin{aligned} \texttt{mult(m, n)} &= \texttt{if (n == 0) 0 else m + mult(m, n - 1)} \\ &= \texttt{m + mult(m, n - 1)} \\ &= \texttt{m + (n - 1)m} \\ &= \texttt{mn} \end{aligned}$$

Note that we were able to make this argument purely symbolically using a symbol `n` to stand for any positive number, without knowing what `n` is. The argument is therefore valid for any $\texttt{n} > 0$. $\diamond$

In summary, given a statement P on natural numbers, we can prove $P(n)$ for every natural number n by the following theorem.

Theorem 1 (The Principle of Mathematical Induction). *For all natural numbers n, let $P(n)$ be a statement. If*

1. *$P(0)$ holds, and*
2. *If $P(n)$ holds, then $P(n+1)$ holds*

then $P(n)$ holds for all natural numbers.

The principle of mathematical induction says that in order to prove that a property is true of all natural numbers, it suffices to do the following:

1. State what variable n you are doing induction on.
2. Express the property you are trying to prove as a property P of n.
3. *Basis*: Prove $P(0)$ holds.
4. *Induction hypothesis (I.H.)*: Assume that $P(n)$ holds.
5. *Induction step*: Show that $P(n+1)$ holds without any assumption on what n is.

In a proof by induction, we first show that the statement holds for the "smallest" values of our inductively defined set, which for the natural numbers is just 0. This is the basis. We then want to show that the statement holds for an arbitrary $n + 1 > 0$. We do not get to assume anything about n except that it is positive and that the statement holds for n. This part of the argument is called the induction step. The assumption that the statement holds for n is called the induction hypothesis. Once we have done these things, we are done; the principle of mathematical induction allows us to conclude that the property holds for all n. In effect, induction provides us with infinitely many results by doing only a finite amount of work.

Recursion and induction are closely related. Induction is often used to show that a recursive procedure computes the correct answer. We saw one examples of this in Example 1 and will see one more in Example 2. This use of induction relies on equational reasoning (substituing equals for equals) and mathematical induction.

Example 2. Consider the following function `fac` for computing the factorial $n! = n \cdot (n-1) \cdot (n-2) \cdot \cdots \cdot 2 \cdot 1 \cdot 1$ of a number:

```
def fac(n: Int): Int =
  if (n == 0) 1 else n * fac(n - 1)
```

To argue that this program is correct, we will do induction on `n`. The property we need to prove is

$$\texttt{fac(n)} = \texttt{n}!$$

Basis: If `n` = `0` then `fac(n)` = `fac(0)` and

`fac(0)` = `if (0 == 0) 1 else n * fac(0 - 1)`	(def. of `fac`)
= 1	(eval. of conditonal)
= 0!	(def. of !).

Hence, the basis is shown.

Induction step: The induction hypothesis states that `fac(n)` = `n`!. We need to show that `fac(n + 1)` = (`n` + 1)!. As

`fac(n + 1)` = `if ((n + 1) == 0) 1 else (n + 1) * fac((n + 1) - 1)`	(def. of `fac`)
= `(n + 1) * fac((n + 1) - 1)`	(eval. of conditional)
= `(n + 1) * fac(n)`	(property of addition)
= `(n + 1) * n`!	(I.H.)
= `(n + 1)`!	(def. of !)

the property follows by the principle of mathematical induction. ◇

2 Structural Induction

The importance of recursion and induction is not limited to functions defined over the natural numbers. Rather, the concept of mathematical induction over the natural numbers is an instance of the more general notion of structural induction over values of an inductively defined type. The following theorem presents the generalized version of Theorem 1.

Theorem 2 (The Principle of Structural Induction). *For each value t of an inductive data type T, let $P(t)$ be a statement. If*

1. *For all of T's atomic structures t, $P(t)$ holds.*
2. *For all of T's composites structures t, if P holds for the immediate substructures of t, then $P(t)$ holds.*

then $P(t)$ holds for all values of the inductive data type T

Example 3. The natural numbers, viewed as an inductively defined type, may be represented in Scala using the following case class definition:

```
sealed abstract class Nat
case object Zero extends Nat
case class Succ(n: Nat) extends Nat
```

Given `n` ∈ `Nat`, we know that `n` is either `Zero` or `Succ(m)` for some `m` ∈ ℕ. Suppose we would like to prove that a property P holds for every `n` ∈ `Nat`: we first prove that P(`Zero`) holds, and then we prove that assuming P(`m`) then P(`Succ`(m)) for every `m` ∈ `Nat`n.

Consider the following recursive program for addition and multiplication of natural numbers, represented as an inductively defined type:

```
def add(n: Nat, m: Nat) = n match {
  case Zero => m
  case Succ(p) => Succ(add(p, m))
}
```

To show that for all `n` $\in$ `Nat`

$$\texttt{add(n, Zero)} = \texttt{n}$$

we will do induction on n.

Basis: For the basis, assume `n` = `Zero`. We need to show that `add(Zero, Zero)` = `Zero`. This follow directly from the definition of `add`.

Induction step: Assume `n` = `Succ(p)` and the induction hypothesis `add(p, Zero)` = `p`. We need to show that `add(Succ(p), Zero)` = `Succ(p)`. As

$$\begin{aligned} \texttt{add(Succ(p), Zero)} &= \texttt{Succ(add(p, Zero))} && \text{(def. of add)} \\ &= \texttt{Succ(p)} && \text{(I.H.)} \end{aligned}$$

the property follows by the principle of structural induction. ◇

Example 4. Lists of integers, viewed as an inductively defined type, may be represented in Scala using the following case class definition:

```
sealed abstract class IntList
case object Nil extends IntList
case class Cons(x: Int, xs: IntList) extends IntList
```

Consider the following recursive functions for concatenating lists and determining the length of lists:

```
def concat(xs: IntList, ys: IntList): IntList = xs match {
  case Nil => ys
  case Cons(z, zs) => Cons(z, concat(zs, ys))
}

def length(xs: IntList): Int = xs match {
  case Nil => 0
  case Cons(_, ys) => 1 + length(ys)
}
```

To show that the `length` function distributes over `concat` we need to show that

$$\texttt{length(concat(xs, ys))} = \texttt{length(xs)} + \texttt{length(ys)}$$

for all `IntLists` `xs` and `ys`. The proof goes by structural induction on xs.

Basis: Assume `xs` = `Nil`. As

`length(concat(xs, ys))`	= `length(concat(Nil, ys))`	(from `xs` = `Nil`)
	= `length(ys)`	(def. of `concat`)
	= `0 + length(ys)`	(property of +)
	= `length(xs) + length(ys)`	(def. of `length`)

the basis is shown.

Induction step: Assume xs = `Cons(z, zs)` and the induction hypothesis `length(concat(zs, ys))` = `length(zs) + length(ys)`. As

`length(concat(Cons(z, zs), ys))`	= `length(Cons(z, concat(zs, ys)))`	(def. of `concat`)
	= `1 + length(concat(zs, ys))`	(def. of `length`)
	= `1 + length(zs) + length(ys)`	(I.H.)
	= `length(Cons(z, zs)) + length(ys)`	(def. of `length`)

the property follows by the principle of structural induction. ◇