Introduction to Functional Programming: Lesson 6

PROOF BY INDUCTION AND ITS RELATION TO RECURSIVE FUNCTIONS

Mathematical induction (or proof by mathematical induction) is a proof technique which is used to prove that a statement (formula, theorem) is true for any "well-ordered set." Most commonly, it is used to establish statements for the set of natural numbers, i.e., for the integers 0, 1, 2, 3, ...

Remark: Do the natural numbers start with 0, or with 1? In middle school mathematics, we start with 1, whereas in computer science, we almost always start with 0. In other words, the first natural number is 0 (which might sound odd, so sometimes we may use a new, non-standard term: "zero-th").

Remark: For our purposes, it is enough to know the meaning of "totally ordered" instead of "well-ordered:" a totally ordered set is well-ordered and a little more. Formally, a *totally ordered set* is one for which there is an ordering relation like "<" or "less than" for integers. Given two distinct integers `m` and `n`, either `m<n` or `n<m`. If we accept that the natural numbers `0`, `1`, `2`, `3`, ... are indeed totally ordered, we need not get into the precise meaning of "well-ordered" and "totally ordered". We will be proving propositions about the natural numbers or about sets whose elements are labeled by the natural numbers.

Let `P(n)` be a well-formed proposition (i.e., an expression that evaluates to `True` or `False` but not both) that involves `n`, a natural number. We want to show that `P(n)` holds, i.e., `P(n)` is true, for all `n`.

Example of `P(n)`: "The sum `0+1+2+ ... +(n-1)+n` equals `n*(n+1)/2`."
In programming, `P(n)` will typically be a statement to the effect that a certain algorithm works as intended or claimed, where `n` refers to the size of the data, e.g., an algorithm correctly sorts a set of `n` values.

There are two cases in a proof by induction:

1. Base case (for small values of `n`, like `n = 0`, or `n = 0` and `1`, or `n = 0`, `1` and `2`, etc.), and
2. Induction case (for larger values of `n` beyond the base case).

The base case is often proved by inspection, e.g., for the example proposition `P(n)` above, `P(0)` is "`0` equals `0*1/2`", or `P(1)` is "`0+1` equals `1*2/2`," which are seen to be true by inspection.

The induction case has two parts: (2.1) the induction hypothesis, and (2.2) the induction step.
(Def: Here, a *hypothesis* is a proposition made as a basis for reasoning, without any assumption of its truth.)

2.1 The induction hypothesis is that `P(n)` holds for some `n >= 0`.

2.2 The induction step is, given that `P(n)` holds, show that `P(n+1)` also holds.

Thus the induction case amounts to `P(n)=>P(n+1)` for all `n >= 0`, where `=>` means "implies."

In our example, `P(n)` is "The sum `0+1+2+ ... +(n-1)+ n` equals `n*(n+1)/2`."
while `P(n+1)` is "The sum `0+1+2+ ... + n + (n+1)` equals `(n+1)*(n+2)/2`."

The induction hypothesis is that `P(n)` holds, i.e., "the sum `0+1+2+ ... +(n-1)+n` equals `n*(n+1)/2`. is true"

We need to show that `P(n+1)`, i.e., "the sum `0+1+2+...+ n +(n+1)` equals `(n+1)*(n+2)/2`" also holds.

Note that the left hand side of `P(n+1)` is the sum `( 0+1+2+ ... + (n-1) + n ) + (n+1)`

which equals `n*(n+1)/2 + (n+1)`	by the induction hypothesis
which equals `(n+1)* (n/2 + 1)`	by factorisation
which equals `(n+1)*(n+2)/2`	by simplification
which is the right hand side of `P(n+1)`	

KMIT R&D Lab
Intro to Func Prog
31 July 2018
rev 26 Aug 2018

Thus assuming `P(n)`, we have shown `P(n+1)`, or we have shown that `P(n) => P(n+1)`

... contd. p. 2

Alternately, the induction hypothesis is to assume that `P(n-1)` holds for some `n>0`, and the induction step is, given that `P(n-1)` holds for some `n>0`, to show that `P(n)` also holds, i.e., to show that `P(n-1) => P(n)`.

Exercise: Assume that `P(n)` is the proposition that "the sum `0+1+2+ ... +(n-1)+n` equals `n*(n+1)/2`" For the induction case, assume `P(n-1)` for `n>0` and show `P(n)`, i.e., show that `P(n-1) => P(n)`

Exercise: Prove that the sum of squares from `0` to `n`, i.e., $0^2+1^2+2^2+ \ldots + n^2 = n*(n+1)*(2n+1)/6$

Significance of Proof by Induction

Suppose we want to show that, say, `P(283)` holds, for some proposition `P(n)`. Then:

First we show that the base case holds—in the simplest form, P(0) holds.

Assume we have proved the induction case, that `P(n) => P(n+1)` holds. Treat this implication as a crank handle. Given `P(0)`, turn the handle once, twice, three times, to show `P(1)`, `P(2)`, `P(3)`. Keep on turning the handle. Turning it `281`, `282`, `283` times lets us show `P(281)`, `P(282)`, and finally, `P(283)` holds.

Computing the sum `0+1+2 ... +n` in Gofer

Two syntactically correct functions are given below to compute the sum. `sumupto1` uses the implication `P(n-1) => P(n)`, while `sumupto2` uses `P(n) => P(n+1)`.

Cut-paste the two functions below into a Gofer script, load them into the interpreter, and verify that both work correctly, for the base case, i.e., argument value `0`, as well as the induction case, e.g., argument value 5.

```
sumupto1 : Int -> Int                  sumupto2 : Int -> Int
sumupto1 . 0 = 0                       sumupto2 . 0 = 0
sumupto1 . n = sumupto1 . (n-1) + n    sumupto2 . (n+1) = sumupto2 . n + n+1
```

Remark (Strong and weak induction): To show `P(n)`, we have assumed `P(n-1)`. Sometimes we need a different hypothesis: to show `P(n)`, the hypothesis will be that `P(i)` holds for all `i` such that `0<=i<=n-1`.
(It can be easily proved that the two hypotheses are equivalent, i.e., the one holds whenever the other holds.)

Remark (Divide and conquer method): Computing problems are often solved by breaking them up into two subproblems of half the size, obtaining their respective solutions using strong induction, and combining the solutions. (Each half-size problem would itself be broken up into two quarter-size problems and solved, etc.)

Example (Algorithm to sort a list of numbers): Break the list into two halves, sort the two half-size lists (using this same algorithm, i.e., break each half-size list into two quarter size lists, etc.), and finally, merge the half-size solutions into a fully sorted list. Note: If the list is of odd length, one of the parts will have one more element than the other. The basis case is that a list of size 0 or 1 (empty or singleton list) is sorted by default.

The proposition `P(n)` is that the algorithm correctly solves a problem of size `n` (i.e., sorts a list of size n), the strong hypothesis is that the algorithm "recursively" sorts lists of half the size (size `n/2`), and the induction step consists of proving that merging the two sorted lists of size `n/2` gives a fully sorted list of size `n`.

This mention of "recursive" directly leads to a discussion of the connection between recursion and induction.
The main difference is that induction goes from the small to the large, whereas it is the other way around in recursion.

Note to students: On the next page, recursive algorithms are discussed. Now it is felt, both by students and by teachers, that recursive programs are "hard" to understand. The designers of this course do not at all agree!

To avoid this misunderstanding, you, the student, have to do two things: (1) understand the basic concept of proof by induction, and (2) understand the connection between recursive algorithms and proof by induction.

Induction and Recursion in programming

Inductive reasoning moves from specific observations to broad generalizations, e.g., proof by induction starts with a base case, an induction hypothesis and an induction step, and proves some proposition for all natural numbers.

An important example of inductive generalization is the Peano axioms for defining `N`, the set of natural numbers:

1. `0` is a natural number
2. If `n` is a natural number, so is `n + 1`
3. Only numbers obtained from rules 1 and 2 are natural numbers.

Here we don't already have a set of natural numbers. We are describing a certain way to characterize which numbers are "natural numbers". Generally, induction is used more in mathematics than in computing.

It does matter that, in defining the natural numbers `N`, we have to start with `0` and "work up". It doesn't work to try define the set `N` by saying that `n` is a natural number if `n−1` is a natural number. But once we have an inductive definition of `N` we can leverage that for other purposes.

Recursion is a concept that is used in programming, and especially, in functional programming.

A *recursive* entity, it is said, is one that is defined in terms of itself. This appears to be a contradiction in terms, for when I give the meaning of a word in a dictionary, the description cannot itself contain the word being described, but this seems to be the case with recursive definition. But this contradiction is only apparent, not real.

From Wikipedia: "Recursion in computer science is a method where the solution to a problem depends on solutions to <u>*smaller instances of the same problem*</u> [emphasis added]. For instance, the definition of n! in terms of (n-1)! is a model example of recursion and proof by induction. The approach can be applied to many types of problems, and is one of the central ideas of computer science."

From Stack Exchange (one person's view): The terms "recursive definition" and "inductive definition" are both quite common, and the differences between the terms are usually so insignificant that either one will work. Some people are fastidious about whether they call each definition "inductive" or "recursive", but I cannot immediately think of an example where changing from one word to the other would change what is going on in a particular definition.

Inductive definitions are more common when we are defining a set of objects "out of nothing", while recursive definitions are more common when we are defining a function on an already-existing collection of objects.

The following definition of the factorial function is a recursive definition:

```
n! = 1, if n=0
     n*(n−1)!, if n>0
```

The correctness of recursive definitions like the one above is shown by employing proof by induction.

Inductive definitions are particularly important in computer science, mathematical logic, and related areas. One additional aspect of them is that they usually give a set of "terms" for the objects being described. For example, we know from the inductive definition of `N` that every natural number can be expressed as a finite string of the form `0+1+1+⋯1+1+1+⋯1` (of the appropriate length). Using this idea, just to develop our recursive programming skills, in practice problems in later lessons, we will define addition of two integers as repeated addition by 1, and likewise, define multiplication of two integers as repeated addition and exponentiation as repeated multiplication.

[Note for students: Many of the descriptions here are adapted from `cseducators.stackexchange.com`. Stack Exchange is a network of question-and-answer (Q&A) websites on topics in diverse fields, each site covering a specific topic, where questions, answers, and users are subject to a reputation award process. As of August 2019, the three most actively-viewed sites in the network are Stack Overflow (for programmers), Super User (for enthusiasts and powerusers) and Ask Ubuntu (for Ubuntu users and developers).]

♋ ♦ ♋

PROOF BY STRUCTURAL INDUCTION, FOR LISTS

(Since you have not seen the *cons* operator `::` for lists, wait for a few Lessons)

Remark: Structural induction, a generalisation of mathematical induction, is used to prove that some proposition `P(x)` holds for all instances `x` of some “recursively defined” structure, such as lists or trees. Often, such a proof proceeds by natural induction on the size of the structure.

Let `P(s)` be a well-formed proposition that involves `s`, a list (in the sense of purely functional programming). We want to show that `P(s)` holds, i.e., `P(s)` is true, for all lists `s` (in a given context, e.g., list type, properties).

The base case will be for small-sized lists, typically the empty list `[]`, and sometimes the singleton list `[x]`. As with natural numbers, the base case for lists is usually shown to be correct by inspection. The induction case will be for non-empty lists, denoted by the pattern `x::xs`, or lists with >2 elements `x::y::ys`. The induction hypothesis is that `P(xs)` is true, and the induction step is to do “something” with x, the list head, and the value of the recursive call on the list tail `xs`, and conclude that `P(x::xs)` holds, i.e., show that `P(xs)=>P(x::xs)`.

```
len: [a] -> Int
len . [ ] = 0
len . (x::xs) = 1 + len . xs
```

```
sum: Num.a => [a] -> Int–all numeric types a
sum . [ ] = 0
sum . (x::xs) = x + sum . xs
```

Exercise: Prove by induction that `len` computes the length of a list, and `sum` computes the sum of list elements.

Proof: Base case is correct by inspection. The induction hypothesis is that `len.xs` is the length of the tail of the list, and `sum.xs` is its sum. The length of a list is 1more than the length of its tail (induction step of `len`), and the sum of a list is got by adding the head to the sum of the tail (induction step of `sum`).

♋ ♦ ♋