1 Ordinary Induction

(Ordinary) mathematical induction is a way to prove things about natural numbers. We can write it as a rule:

$$\frac{\phi(0) \quad \forall x.\, \phi(x) \supset \phi(succ(x))}{\forall x.\, \phi(x)}$$

where $\phi(x)$ is a statement involving a number x. Proving $\phi(0)$ is called the *base* case, and proving that $\phi(succ(x))$ follows from $\phi(x)$, for any x, is called the *induction step (or case).*

There is a good source of examples of mathematical induction in (what are known as) primitive recursive definitions. Here is an example, we define addition by the following two equations:

$$\begin{aligned}&0 + y = y\\&succ(x) + y = succ(x + y)\end{aligned}$$

where $succ(x)$ is the next number after x, its successor.

Note that these equations completely determine addition. The first one tells you how to add 0 to something and the second one tells you how to add the successor of something to something else, provided you know how to add the something to the something else. Looking at them as recursive definitions, we say that the recursion is on the first argument.

Using mathematical induction we can then prove the *associativity* of addition, which is this statement:

$$\forall x, y, z.\, x + (y + z) = (x + y) + z$$

Here is the proof, which we write out in detail. We take $\phi(x)$ to be:

$$\forall y, z.\, x + (y + z) = (x + y) + z$$

The first thing we have to do is the base case, i.e., we have to prove $\phi(0)$ which is:

$$\forall y, z.\, 0 + (y + z) = (0 + y) + z$$

To this end we choose y and z and we have to prove:

$$0 + (y + z) = (0 + y) + z$$

To show this we first work on the left-hand-side of the equation:

$$0 + (y + z) \quad = \quad (y + z) \quad \text{(by the definition of +)}$$

and then on the right-hand-side:

$$(0 + y) + z \quad = \quad (y + z) \quad \text{(again, by the definition of +)}$$

and we are done with the base case as we have got to the same place.

Now we have to do the induction step. So we choose a number x, assume

$$\forall y, z.\, x + (y + z) = (x + y) + z$$

(this is called the *induction hypothesis*) and we have to prove that

$$\forall x, y, z.\, \mathit{succ}(x) + (y + z) = (\mathit{succ}(x) + y) + z$$

So we choose y and z and we now have to prove:

$$\mathit{succ}(x) + (y + z) = (\mathit{succ}(x) + y) + z$$

We again do this by working on both sides of the equation. Starting on the left:

$$\mathit{succ}(x) + (y + z) \quad = \quad \mathit{succ}(x + (y + z)) \quad \text{(by the definition of +)}$$

and on the right:

$$\begin{array}{rcll} (\mathit{succ}(x) + y) + z & = & \mathit{succ}(x + y) + z & \text{(by the definition of +)} \\ & = & \mathit{succ}((x + y) + z) & \text{(again, by the definition of +)} \end{array}$$

So we still have to prove:

$$\mathit{succ}(x + (y + z)) = \mathit{succ}((x + y) + z)$$

but that follows because we know from the induction hypothesis (instantiating the universal quantifiers) that:

$$x + (y + z) = (x + y) + z$$

In this example we did induction on x. We could also have tried induction on y when we would have taken $\phi(y)$ to be:

$$\forall x, z.\, x + (y + z) = (x + y) + z$$

But that would not have worked so well. The reason is that the definition of $+$ is on the first of its two arguments, so if we look at

$$x + (y + z)$$

the definition "works" on x, and if we look at

$$(x + y) + z)$$

the definition works on $x + y$ and there it works on x. So x comes up on both sides of the equation and is therefore the natural choice of *induction variable.*

Here is an example you can try:

$$\text{Commutativity} \quad \forall x, y. x + y = y + x$$

This one is hard since either of x or y are equally natural as induction variables, one from looking at the left and the other from looking at the right. In fact whichever one you choose in doing the base case (and also in doing the induction step) you will need to prove something about the other which will require another induction. But it works in the end!

To get more examples one can define multiplication and exponentiation by:

$$\begin{aligned} &0 \times y = 0 \\ &succ(x) \times y = y + (x \times y) \end{aligned}$$

and

$$\begin{aligned} &y^0 = 1 \\ &y^{succ(x)} = y \times (y^x) \end{aligned}$$

and there are lots of things for you to prove:

$$\begin{aligned} &\forall x.\, x \times 1 = x \\ &\forall x, y, z.\, x \times (y + z) = (x \times y) + (x \times z) \\ &\forall x, y.\, (x \times y) = (y \times x) \\ &\forall x, y, z.\, x \times (y \times z) = (x \times y) \times z \\ &\forall x, y, z.\, (y \times z)^x = y^x \times z^x \\ &\forall x, y, z.\, y^{x \times z} = (y^z)^x \\ &\forall x, y, z.\, y^{x+z} = y^x \times y^z \end{aligned}$$

The order they are written in here is deliberate: you will find that, to prove later ones in the list, you will sometimes need earlier ones; you will also sometimes need the properties of addition, $+$, discussed above. The third example is the hardest; if you can't do it, just go on to the others.

2 List Induction

Here we consider lists (same thing as strings) over an alphabet Σ; that is we are interested in Σ^* rather than the natural numbers. So what now is the right kind of induction principle to prove things about all lists? Well natural numbers are built up from 0 by continually taking successors: you get 0,1,2,3, etc.

In the same way lists are built up from the empty list ϵ by adding on elements of Σ at the front. For example, take the case where Σ has just two elements a and b. Then starting from ϵ we first get a and b; then we get aa and bb and also ab and bb. The next time round we get eight things, starting at aaa and finishing at bbb. The function which adds an element of Σ to the beginning of a list is called *cons* and is written with a raised dot, thus: $l \cdot x$, so that, for example, $b \cdot aa$ is baa.

So the natural induction principle says that if however you build up your list what you want to prove remains true, then it is true for all lists. In symbols we have the following *principle of list induction* for Σ^*:

$$\frac{\phi(\epsilon) \quad \forall l, x.\, \phi(x) \supset \phi(l \cdot x)}{\forall x.\, \phi(x)}$$

Once we have the empty list and consing we can define other functions by the list version of primitive recursion. For example list concatenation is called *append* and it is defined by:

$$\begin{aligned}&\mathit{append}(\epsilon, y) = y\\&\mathit{append}(l.x, y) = l.\mathit{append}(x, y) \quad \text{(for } l \text{ in } \Sigma)\end{aligned}$$

(In CS2 we wrote $\mathit{append}(x, y)$ simply as xy, but here it helps to have the append function made "visible.")

Then a natural thing to try to prove by list induction is again associativity, but of append this time:

$$\forall x, y, z.\, \mathit{append}(x, \mathit{append}(y, z)) = \mathit{append}(\mathit{append}(x, y), z)$$

It turns out that this is very similar indeed to the situation with addition. It is again natural to try the induction formula:

$$\forall y, z.\, \mathit{append}(x, \mathit{append}(y, z)) = \mathit{append}(\mathit{append}(x, y), z)$$

since both sides of the equation work on x. The base case is:

$$\forall y, z.\, \mathit{append}(\epsilon, \mathit{append}(y, z)) = \mathit{append}(\mathit{append}(\epsilon, y), z)$$

and is left to you to do as an exercise (a rather small one!).

For the induction step, we assume:

$$\forall y, z.\, \mathit{append}(x, \mathit{append}(y, z)) = \mathit{append}(\mathit{append}(x, y), z)$$

and we have to prove, for any l in Σ, that:

$$\forall y, z.\, \mathit{append}(l \cdot x, \mathit{append}(y, z)) = \mathit{append}(\mathit{append}(l \cdot x, y), z)$$

To that end we choose y and z and we now have to prove:

$$\mathit{append}(l \cdot x, \mathit{append}(y, z)) = \mathit{append}(\mathit{append}(l \cdot x, y), z)$$

To do this we work on both sides of the equation, just like before, but now using the definition of append: the left-hand-side reduces to:

$$l \cdot \mathit{append}(x, \mathit{append}(y, z))$$

and the right-hand-side reduces to:

$$l \cdot \mathit{append}(\mathit{append}(x, y), z)$$

and we use the induction hypothesis and we are done. You might like to fill in the details here—they are just as in the case of addition.

The reverse function does as its name suggests, so that, for example:

$$\mathit{rev}(abc) = bca$$

It can be defined by:

$$\begin{aligned}&rev(\epsilon) = \epsilon\\&rev(l.x) = append(rev(x), l) \quad \text{(for } l \text{ in } \Sigma\text{)}\end{aligned}$$

where l is really a short way of writing $l \cdot \epsilon$: it is an element of Σ thought of as a (short) list.

Here are a couple of things that can be proved by list induction (exercises).

$$\forall x, y.\, rev(append(x, y)) = append(rev(y), rev(x))$$

$$\forall x.\, rev(rev(x)) = x$$

3 Tree Induction

Lists are a generalisation of numbers, as numbers can be thought of as lists over an alphabet Σ with a single element, say 1. The idea is that 0 corresponds to ϵ and $succ(x)$ corresponds to $l \cdot x$, so that, for example $3 = succ(succ(succ(0)))$ corresponds to $111 = 1 \cdot 1 \cdot 1 \cdot \epsilon$. One then sees that ordinary induction and primitive recursion correspond to list induction and primitive recursion for these "unary" lists.

We now see that in just the same way trees generalise lists. Trees are also built from alphabets Σ, but the alphabets are *graded*, meaning that each element of the alphabet has a *rank*, also called an *arity*, given by a function

$$\text{rank} : \Sigma \to \mathcal{N}$$

The elements of the alphabet Σ are called *letters* or *function symbols*.

Given that, the Σ-*trees* are defined by saying that if $t_1, \ldots, t_n$ are Σ-trees and a is a letter of rank n then:
(tree picture missing here: it has a tip labelled by a and n branches pointing to $t_1 \ldots t_n$)
is a Σ-tree.

One can equivalently talk about Σ-*terms*, which are defined by saying that if $t_1, \ldots, t_n$ are Σ-terms and a is a letter of rank n then $a(t_1, \ldots, t_n)$ is a Σ-term. Such terms can be alternatively defined by a context-free grammar with a nonterminal T and a production of the form:

$$T \to a(T, \ldots, T)$$

for each letter a of arity n, where there are n T's. (If you have done 1st-order logic you will see that these are exactly the terms of first-order logic with the given function symbols containing no variables.) Usually we know what Σ is and we just talk of trees or terms, rather than Σ-trees or Σ-terms.

Here are some examples.

Natural numbers Here

$$\Sigma = \{\text{succ}:1, \text{zero}:0\}$$

where we have shown both the letters and their arity. Here the context-free grammar is:

$$T \to \text{succ}(T) \mid \text{zero}()$$

The possible numbers in this sense are zero(), succ(zero()), succ(succ(zero())), etc, evidently corresponding to $0, 1, 2$ etc.

Lists (of a's and b's) Here:

$$\Sigma = \{a\!:\!1, b\!:\!1, \text{NIL}\!:\!0\}$$

Here the context-free grammar is:

$$T \to s(T) \mid b(T) \mid \text{NIL}()$$

The possible lists in this sense are NIL, $a(\text{NIL}())$, $b(\text{NIL}())$, $a(a(\text{NIL}()))$, $a(b(\text{NIL}()))$, $b(a(\text{NIL}()))$, etc, corresponding to $\epsilon, a, b, aa, ab, ba$ etc.

Binary Trees (with a and b at their leaves.Here:

$$\Sigma = \{P\!:\!2, a\!:\!0, b\!:\!0,\}$$

Here the context-free grammar is:

$$T \to P(T, T) \mid a() \mid b()$$

Let us consider the substitution $sub_a(t, u)$ of one binary tree u for all occurrences of a in another binary tree t. For example:

$$sub_a(P(P(b, a), a), P(b, a)) = P(P(b, P(b, a)), P(b, a))$$

This can be defined using a tree form of primitive recursion:

$$\begin{array}{lcc} sub_a(P(t_1, t_2), u) & = & P(sub_a(t_1, u), sub_a(t_2, u)) \\ sub_a(a(), u) & = & u \\ sub_a(b(), u) & = & b() \end{array}$$

Now suppose that we want to prove the following:

$$\forall t, u, v.\, sub_a(sub_a(t, u), v) = sub_a(t, sub_a(u, v))$$

To do this we are going to use the principle of *tree induction*, also called *structural induction*, here for binary trees:

$$\frac{\forall t_1, t_2, .\, \phi(t_1) \wedge \phi(t_2) \supset \phi(P(t_1, t_2)) \quad \phi(a()) \quad \phi(b())}{\forall t.\, \phi(t)}$$

In this case we are going to take the induction formula $\phi(t)$ to be:

$$\forall u, v.\, sub_a(sub_a(t, u), v) = sub_a(t, sub_a(u, v))$$

So, we have first to prove that $\phi(P(t_1, t_2))$, assuming that $\phi(t_1)$ and $\phi(t_2)$. So fixing u and v we have to prove that:

$$sub_a(sub_a(P(t_1, t_2), u), v) = sub_a(P(t_1, t_2), sub_a(u, v))$$

starting on the left, we calculate:

$$\begin{array}{lcll} sub_a(sub_a(P(t_1,t_2),u),v) & = & sub_a(P(sub_a(t_1,u),sub_a(t_2,u)),v) & \text{(by the definition of } sub_a\text{)} \\ & = & P(sub_a(sub_a(t_1,u),v),sub_a(sub_a(t_2,u),v)) & \text{(by the definition of } sub_a\text{)} \\ & = & P(sub_a(t_1,sub_a(u,v)),sub_a(t_2,sub_a(u,v))) & \text{(by induction hypothesis)} \end{array}$$

and, now starting on the right, we calculate:

$$\begin{array}{lcll} sub_a(P(t_1,t_2),sub_a(u,v)) & = & P(sub_a(t_1,sub_a(u,v)),sub_a(t_2,sub_a(u,v))) & \text{(by the definition of } sub_a\text{)} \end{array}$$

Next we have to prove that $\phi(a())$. So fixing u and v we have to prove that:

$$sub_a(sub_a(a(),u),v) = sub_a(a(),sub_a(u,v))$$

which is easy as both sides work out to $sub_a(u,v)$, using the definition of sub_a.

Finally we have to prove that $\phi(b())$. So fixing u and v we have to prove that:

$$sub_a(sub_a(b(),u),v) = sub_a(b(),sub_a(u,v))$$

which is also easy as both sides work out to $b()$, again using the definition of sub_a.

We can make use things more readable by using the notation:

$$t[u/a]$$

for $sub_a(t,u)$. Then we can write:

$$\forall t,u,v.\, sub_a(sub_a(t,u),v) = sub_a(t,sub_a(u,v))$$

as:

$$\forall t,u,v.\, t[u/a][v/a] = t[u[v/a]/a]$$

Try rewriting the above proof using this notation: it should look a lot clearer! Here is another such "law of substitution:"

$$\forall t,u.\, t[u/a] = t \text{ (if } a \text{ is not in } t\text{)}$$

Try proving it by tree induction; show too that the condition that a is not in t is needed, in the sense that if you remove it, then the statement becomes false.

Obviously, one can also define $sub_b(t,u)$, and use the corresponding notation $t[u/b]$. Then, what happens if we do a substitution on, say, a, followed by a substitution on b? We get the following law:

$$\forall t,u,v.\, t[u/a][v/b] = t[v/b][u[v/b]/a] \text{ (if } a \text{ is not in } v\text{)}$$

Try proving it by tree induction; show too that the condition that a is not in v is needed, in the sense that if you remove it, then the statement becomes false.

Tree induction for binary trees is obviously a special case of tree induction for any graded alphabet Σ. This is as follows: to prove

$$\forall x.\phi(x)$$

where x is ranging over all Σ-trees, one proves

$$\forall x_1, \ldots x_n.\, \phi(x_1) \wedge \ldots \wedge \phi(x_n) \supset \phi(a(x_1, \ldots, x_n))$$

for all letters a, where n is the arity of a. Check that, using the above tree representations of natural numbers and lists, ordinary induction and list induction are special cases of this general form of tree induction.

Abstract syntax

In his notes, Pitts uses *abstract syntax.* For example, LC arithmetic expressions are given by

$$E ::= n \,|!l \mid E\, iop\, E$$

where n ranges over integers and l over locations and iop over arithmetic operations. This should not be read as a context-free grammar but rather as giving a graded alphabet, viz:

$$\Sigma = \{n \mid n \in Z\} \cup \{!l \mid l \in L\} \cup \{iop \mid iop \in Iop\}$$

See Pitts' notes, pp. 4,5. All the n and $!l$ have rank 0 and all the arithmetic operations have rank 2.

One other small point: our "official notation" for trees, viz terms, results in many parentheses. The abstract syntax also gives notational conventions to avoid this, such as, here, using infix to write arithmetic operations (and one should add parentheses or use standard priorities to deal with any resulting ambiguities).

So structural induction (= tree induction) for arithmetic expressions is as follows:

$$\frac{\forall n.\, \phi(n) \qquad \forall l.\, \phi(!l) \qquad \forall E_1, E_2. \forall\, iop.\, \phi(E_1) \wedge \phi(E_2) \supset \phi(E_1\, iop\, E_2)}{\forall E.\, \phi(E)}$$

This is exactly what Pitts has on his p. 13. Note that his leaf nodes correspond to our letters of degree 0