CSCI 2041: Advanced Programming Principles
Proving Program Properties Using Induction

Gopalan Nadathur

This document provides several examples of proofs by induction. Many of these proofs are ones we would have already carried out in class. The main virtues of this document are that (a) it clarifies some of the details if you missed them in class and (b) it gives you a feel for how to write such proofs in homeworks and exams.

As we did in class, we look first at proofs by induction on natural numbers and then consider proofs by structural induction based on type declarations.

1 Proofs by Induction on Natural Numbers

We considered two versions of the induction principle for natural numbers in class, one called weak induction or just induction and the other called strong induction. In both cases, we are interested in showing that some property $P(n)$ holds for all natural numbers n; the collection of natural numbers is usually denoted by N and thus what we are trying to show can be written as $\forall n \in N.P(n)$. The particular methods are then the following:

Principle of (Weak) Induction. To show $\forall n \in N.P(n)$, we carry out the following steps:

1. we show that $P(0)$ holds, and
2. we show that if $P(n)$ holds then $P(n+1)$ holds.

The reason why this works is that by doing this we would have provided a method for showing $P(n)$ for any natural number n: we start with $P(0)$, use what we would have done in step (2) to show $P(1)$ and keep repeating this till we have shown $P(n)$. When we write up proofs based on this approach, we usually label the first step the *base case* and the second step the *inductive step*. Moreover, the assumption $P(n)$ in the inductive step is referred to as the *induction hypothesis*.

Principle of Strong Induction. Here, to show $\forall n \in N.P(n)$, we provide a method for showing that $P(n)$ holds assuming that $P(m)$ holds for every $m < n$. This argument "works" in a similar way to the principle of weak induction: $P(0)$ must hold without any assumptions because there is nothing smaller than 0; then at each step, we would have already shown that $P(m)$ holds for every m smaller than n and so we can use the method we have set out to conclude that $P(n)$ holds; finally we can keep going this way, till we reach the particular n for which we want to show $P(n)$ holds. When we carry out a proof in this style, we refer to the assumption that $P(m)$ holds for all $m < n$ as the induction hypothesis.

You may wonder why in the principle of strong induction you have only *one* thing to show, i.e. don't you need to show $P(0)$ separately? In practice you probably need to split your proof into separate cases, treating $P(0)$ differently from the other cases. However, in the description of the method, this case is subsumed into the others: it is just that when you are showing $P(0)$, you cannot use any assumptions because there is nothing smaller than 0.

Are the weak and strong induction principles really of different strength? In fact not! In other words, the terminology of "weak" and "strong" is a bit misleading. Clearly, weak induction is a

special case of strong induction: we would have to show $P(0)$ without assuming anything about any other number and if you can show that $P(n+1)$ follows from only $P(n)$ then it must also follow from the assumption that $P(m)$ holds for every $m < n$. In the other direction, we make the property we want to prove stronger and then use weak induction instead of strong induction. In particular, instead of showing $\forall n \in N.P(n)$ we show $\forall n \in N.\forall m < n.P(m)$. When we show the latter by weak induction, it will really amount to using strong induction—we assume $\forall m < n.P(m)$ holds, use this to show that $P(n+1)$ holds and then combine it with $\forall m < n.P(m)$ to conclude $\forall m < (n+1).P(m)$ holds. Further, from $\forall n \in N.\forall m < n.P(m)$ we can easily conclude that $P(k)$ holds for any k—if you pick n in what you have shown to be $k+1$, then we get $\forall m < (k+1).P(m)$ which means that $P(k)$ also must be true—and so we have $\forall k \in N.P(k)$.

Thus, which principle you use, whether weak or strong, is only a matter of convenience. A practical matter: how do we decide which one is more convenient? This is usually dependent on the program we want to reason about. If the value of a function at some input number depends only on the input at the previous number then the weak induction principle should be adequate. If it depends on several preceding values, then the strong induction principle may be more useful.

Lets look now at a few concrete examples.

1.1 Correctness of the Iterative Factorial Function.

The iterative (or tail-recursive) version of the factorial function that we considered in class is the following:

```
let tr_fact n =
   let rec fact_helper n a =
     if (n = 0) then a
     else fact_helper (n-1) (n * a)
   in fact_helper n 1
```

Our goal in this case is to show the following

Claim 1. *(tr_fact n) evaluates to $n!$.*

However, as we observed in class, we cannot show this directly: tr_fact is defined using $fact\_helper$ and so we need some property about $fact\_helper$ to be able to prove this. What can this property be? Here we have to balance two requirements:

- the property we choose for $fact\_helper$ has to be strong enough to yield what we want to show about tr_fact, and
- we should also be able to show that the property actually holds for $fact\_helper$.

There is a real tradeoff here and we have to think carefully in coming up with a suitable property. To see this, suppose we pick the following property for tr_fact: for every n and m, (tr_fact n m) evaluates to $n!$. This property is clearly strong enough for us to prove Claim 1—if (tr_fact n m) evaluates to $n!$ for any m, then it must be the case that (tr_fact n 1) evaluates to $n!$. However, it is not a property that tr_fact satisfies: for example, (tr_fact 0 7) evaluates to 7 that is definitely not the factorial of 0.

Coming up with a suitable property for $fact\_helper$ requires some thought, there is no silver bullet. We actually went through this process in class; we tried out a few example calls to $fact\_helper$ to discover a pattern before we converged on a plausible one. At the end of it, we settled on the following lemma that we ended up proving by induction as shown.

Lemma 2. *For all natural numbers n and m, $(fact\_helper\ n\ m)$ evaluates to $n! * m$.*

Proof. We will prove the property we want by induction on n. In other words, we will use natural number induction to show the property $\forall n \in N.P(n)$ where $P(n)$ is the following property:

$$\forall m \in N.(fact\_helper\ n\ m) \text{ evaluates to } n! * m.$$

We work out the base case and the inductive step below.

Base Case: What we need to show here is $P(0)$, i.e.

$$\forall m \in N.(fact\_helper\ 0\ m) \text{ evaluates to } 0! * m.$$

To show this, it is enough to show for any arbitrary m that $(fact\_helper\ 0\ m)$ evaluates to 0!. Observe that 0! is 1 and so $0! * m$ is the same as m. An inspection of the definition of $fact\_helper$ shows that $(fact\_helper\ 0\ m)$ evaluates to m, concluding the proof of this case.

Inductive Step: The induction hypothesis tells us that $P(n)$, i.e.

$$\forall m \in N.(fact\_helper\ n\ m) \text{ evaluates to } n! * m,$$

holds. What we have to show based on this is that $P(n+1)$, i.e.

$$\forall m \in N.(fact\_helper\ (n+1)\ m) \text{ evaluates to } (n+1)! * m$$

holds. To show this, it is enough to show that $(fact\_helper\ (n+1)\ m)$ evaluates to $(n+1)! * m$ for any arbitrary m. Looking at the definition of $fact\_helper$, we see that $(fact\_helper\ (n+1)\ m)$ evaluates to whatever $(fact\_helper\ n\ ((n+1) * m))$ evaluates to. Plugging in in $(n+1) * m$ for the quantified variable in the induction hypothesis, we see that this must be $n! * (n+1) * m$. Rearranging this expression we see that it is the same as $(n+1)! * m$ as desired. □

The lemma above is really the hard part. Once we have proved it, we can use it to easily conclude that Claim 1 must be true. This part of the argument is left as an exercise.

1.2 Correctness of the Simple Fibonnaci Function

The simple version of the Fibonnaci function that we defined in class is the following:

```
let rec fib n =
   if (n = 1) then 1
   else if (n = 2) then 1
        else fib (n-1) + fib (n-2)
```

To state the property we wanted of it, we first had to identify the mathematical function it is supposed to emulate:

$$math\_fib(n) = \begin{cases} 1 & \text{if } n = 1 \\ 1 & \text{if } n = 2 \\ math\_fib(n-1) + math\_fib(n-2) & \text{otherwise} \end{cases}$$

Then the property we want to prove about the OCaml definition of the Fibonnaci function is the following:

Theorem 3. *For all natural numbers $n > 0$, $(fib\ n)$ evaluates to $math\_fib(n)$.*

Proof. Observe here that $(fib\ n)$ for $n > 2$ is defined in terms of $(fib\ (n-1))$ and $(fib\ (n-2))$. As explained earlier, this is a situation where we would find it advantageous to use strong induction. Using this principle, we will prove $\forall n.(n > 0)$ implies $P(n)$, where $P(n)$ is the following property:

$$(fib\ n) \text{ evaluates to } math\_fib(n).$$

Recalling the method, we get as an assumption the following for $n > 0$:

$$\forall m < n.(m > 0) \text{ implies } (fib\ m) \text{ evaluates to } math\_fib(m).$$

What we need to conclude from this induction hypothesis is

$$(n > 0) \text{ implies } (fib\ n) \text{ evaluates to } math\_fib(n).$$

As is generally the case, the proof here splits into a few subcases. We show $P(1)$ and $P(2)$ first, without using the induction hypothesis. For example, to conclude $P(1)$ we have to show that

$$(fib\ 1) \text{ evaluates to } math\_fib(1).$$

But this follows immediately from an inspection of the definition of fib for the case that its argument is 1. In the case that $n > 2$, $(fib\ n)$ evaluates to whatever $(fib\ (n-1) + (fib\ (n-2)$ evaluates to. But using the induction hypothesis, this must be the same as $math\_fib(n-1) + math\_fib(n-2)$, which is, by definition, $mat\_fib(n)$ as required. □

2 Proofs by Structural Induction Based on Type Declarations

A type declaration identifies a finite set of constructors that completely determine the way any object of that type can be put together. Moreover, this construction process must start from some base cases, using constructors that does not take any arguments or that take only objects of other pre-existing types as arguments, and then proceed by possibly using constructors that use simpler objects of the type in question to yield more complex objects. Just like with natural numbers, the fact that objects are constructed in this disciplined, inductive fashion gives us a weak and a strong induction principle associated with the type declaration. Just like with natural numbers, the principles are based on the idea that we can prove that a given property holds for all objects of the type by providing a method for generating the proof for any object in question. We look at a few examples of the use of this idea below. In these examples, we will use only the weak form of the induction principle but clearly a strong form also makes sense and can be useful if function definitions depend on more than just the objects constructed in the immediately preceding step.

2.1 Induction on a Explicit Representation of Natural Numbers

We considered the following type declaration and functions in class:

```
type nat = Zero | Succ of nat

let rec plusNat x y =
   match x with
   | Zero -> y
   | (Succ x') -> Succ (plusNat x' y)

let rec toInt n =
   match n with
   | Zero -> 0
   | (Succ n') -> toInt n' + 1
```

In this setting, we articulated a correctness property for *plusNat* and proved it using induction on the definition of the *nat* type. The property and the proof are reproduced below in case you did not understand them completely in class.

Theorem 4. *For every n and m of type nat, $toInt\ (plusNat\ n\ m) = (toInt\ n) + (toInt\ m)$.*

Proof. Before we present a proof of the theorem, we should clarify the meaning of the $=$ that is used in its statement. What we mean by this here and anywhere else we use it in positing properties of OCaml functions is that the expressions on both sides of it have the same *termination* behaviour and that they evaluate to the same values when they do terminate. We must be careful to preserve this meaning in every step of the argument we provide for such theorems.

To get to the details, we will prove this theorem by induction on the structure of n which is of type *nat*. Thus, what we will prove is $\forall n \in nat.P(n)$, where $P(n)$ is the property

$$\text{for all } m \in nat.\ toInt\ (plusNat\ n\ m) = (toInt\ n) + (toInt\ m).$$

The proof will break up into a base case and an inductive step.

Base Case. Here we want to show $P(Zero)$, which is

$$\text{for all } m \in nat.\ toInt\ (plusNat\ Zero\ m) = (toInt\ Zero) + (toInt\ m).$$

We will pick m to be arbitrary and show that the equality holds; since m was chosen to be arbitrary, the equality will then obviously have to hold for *all* m. Noting that the argument of a function is evaluated first and using the definition of $plusNat$, we see that the lefthand side will evaluate to whatever $(toInt\ m)$ evaluates to. In a similar fashion, we see that the righthand side also evaluates to whatever $(toInt\ m)$ evaluates to. Thus the equality stands verified.

Inductive Step. Here we get to assume that

$$\text{for all } m \in nat.\ toInt\ (plusNat\ n\ m) = (toInt\ n) + (toInt\ m)$$

holds and we need to show that

$$\text{for all } m \in nat.\ toInt\ (plusNat\ (Succ\ n)\ m) = (toInt\ (Succ\ n)) + (toInt\ m)$$

must also hold. We pick m to be arbitrary and show that

$$toInt\ (plusNat\ (Succ\ n)\ m) = (toInt\ (Succ\ n)) + (toInt\ m);$$

from this it will follow that the equality holds for *any* m. Again observing that the argument of a function must be evaluated first, and using the definitions of $toInt$ and $plusNat$, we can conclude that the lefthand side of what we have to prove will evaluate to whatever $(toInt\ (plusNat\ n\ m)) + 1$ evaluates to. Similarly, we can conclude that the righthand side will evaluate to whatever $(toInt\ n) + 1 + (toInt\ m)$ will evaluate to. Rearranging the latter to $(toInt\ n) + (toInt\ m) + 1$ and using the induction hypothesis, we see that the righthand side must also evaluate to whatever $(toInt\ (plusNat\ n\ m)) + 1$ evaluates to. But then the equation obviously stands verified. □

2.2 Structural Induction on Lists

Lists are built in in OCaml. However, as we have observed in class, we can view this type as being given by the following type declaration:

```
type 'a list = [] | :: of 'a * 'a list
```

This yields an induction principle for lists that we have articulated explicitly in class and that you can consult in the lecture slides. We will use this principle in what follows.

In class we saw the following function definitions over lists:

```
let rec append l1 l2 =
   match l1 with
   | [] -> l2
   | (h::t) -> h :: (append t l2)

let rec reverse l =
   match l with
   | [] -> []
   | (h::t) -> append (reverse t) [h]
```

Let us try to prove the following property that relates these two functions; in class we will use this property to eventually conclude that *reverse* is involutive, i.e. that if you apply it twice you get back the original list. Note that we use equality again in expressing the property in the theorem. This should be interpreted in the way previously explained: the two sides of the equality have the same termination behaviour and, if they both terminate, then they produce the same value.

Theorem 5. *For all* $l \in (\prime a\ list)$*, for all* $x \in \prime a$*.* $(reverse\ (append\ l\ [x])) = x :: (reverse\ l)$*.*

Proof. There is one further thing to note in the statement of the theorem: we have used a type variable ($\prime a$) in it. What this means is that we implicitly quantify over all possible types. In the proof we will realize this implicit quantification by not using any fact that depends specifically on the value we chose for this type. We have to be careful to follow this principle scrupulously, otherwise we will end up proving incoherent things.

Now, the proof of the theorem will be by induction on the structure of the list l. Thus, what we will prove is $\forall l \in (\prime a\ list).P(l)$, where $P(l)$ is the property

$$\text{for all } x \in \prime a.\ (reverse\ (append\ l\ [x])) = x :: (reverse\ l).$$

The proof splits up into a base case and an inductive step.
Base Case. Here we have to show $P([\,])$, i.e. that

$$\text{for all } x \in {}'a, (reverse\ (append\ [\,]\ [x])) = x :: (reverse\ [\,])$$

holds. We pick x to be arbitrary and show that

$$(reverse\ (append\ [\,]\ [x])) = x :: (reverse\ [\,])$$

from which the desired conclusion follows. Making use of the definitions of *reverse* and *append*, it is easy to see that both the left and the right sides evaluate to $[x]$ and thus the desired equality is verified.
Inductive Step. Here we get to assume that

$$\text{for all } x \in {}'a.\ (reverse\ (append\ l\ [x])) = x :: (reverse\ l)$$

holds and we need to show that

$$\text{for all } x' \in {}'a.\ \text{for all } x \in' a, (reverse\ (append\ (x' :: l)\ [x])) = x :: (reverse\ (x' :: l)).$$

Once again, we pick x' and x arbitrarily and show that

$$(reverse\ (append\ (x' :: l)\ [x])) = x :: (reverse\ (x' :: l)),$$

from which the desired conclusion obviously follows. Using the definitions of *append* and *reverse*, we see that the lefthand side evaluates to whatever

$$(append\ (reverse\ (append\ l\ [x]))\ [x'])$$

evaluates to. Using the induction hypothesis, it follows that this must evaluate to whatever $(append\ (x :: (reverse\ l))\ [x'])$ evaluates to. Using the definition of *append* again, we see that this evaluates to whatever $x :: (append\ (reverse\ l)\ [x'])$ evaluates to. But using the definition of *reverse* on the righthand side, we see that this is exactly what the same for it as well: it must evaluate to whatever $x :: (append\ (reverse\ l)\ [x'])$ evaluates to. □

As an exercise, try writing out a proof of the following fact:

$$\text{for all } l \in ({}'a\ list), (reverse\ (reverse\ l)) = l.$$

You should do this using induction and the fact already proved in Theorem 5.

2.3 Structural Induction on Trees

As a last example, let us consider the definition of the binary tree type that we saw in class:

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

In this context, we defined the following functions

```
let rec insert t i =
   match t with
   | Empty -> Node (i,Empty,Empty)
   | Node (i',l,r) ->
        if (i < i') then Node (i',insert l i,r)
        else Node (i',l,insert r i)

let rec find t i =
   match t with
   | Empty -> false
   | Node (i',l,r) ->
       if (i = i') then true
       else if (i < i') then find l i
            else find r i
```

Both *insert* and $find$ implicitly assume that we are dealing with binary search trees. In class, we will discuss showing that *insert* has been correctly implemented. This will involve proving various properties, some of which we will do in class, some in the lab and one in the homework. In this handout we will prove the remaining piece, which is stated in the following theorem; remember how type variables are to be interpreted in such theorems.

Theorem 6. *For all $t \in ('a\ btree)$, for all $x \in 'a$, if $((insert\ t\ x)$ evaluates to t' then it is the case that, for all $y \in 'a$, if $(find\ t\ y)$ evaluates to true then $(find\ t'\ y)$ evaluates to true.*

Proof. We will prove this by induction on t. What we will show then is $\forall t \in ('a\ btree).P(t)$ where $P(t)$ is the following property:

$\forall x \in 'a.(insert\ t\ x)$ evaluates to t' implies that
$(\forall y \in 'a.$if $(find\ t\ y)$ evaluates to true then $(find\ t'\ y)$ evaluates to true).

As usual, the proof breaks up into an base case and an inductive step.

Base Case. Here we have to show $P(Empty)$, i.e.

$\forall x \in 'a.(insert\ Empty\ x)$ evaluates to t' implies that
$(\forall y \in 'a.$if $(find\ Empty\ y)$ evaluates to true then $(find\ t'\ y)$ evaluates to true).

But this is trivially true, because, looking at the definition of $find$, we see that, no matter how we pick y, $(find\ Empty\ y)$ can never evaluate to true.

Inductive Step. In this case, we assume that t is of the form $Node\ (i, l, r)$ and that $P(l)$ and $P(r)$ are both true. Our goal is to then show that $P(t)$ is true. For this, it suffices to show the following for arbitrary x and y, assuming that $(insert\ (Node\ (i, l, r))\ x)$ evaluate to t':

If $(find\ (Node\ (i, l, r))\ y)$ evaluates to true then $(find\ t'\ y)$ evaluates to true.

The argument splits into two subcases corresponding to whether or not $x < i$.

Subcase $x < i$. In this case t' is whatever $(Node\ (i, insert\ l\ x, r))$ evaluates to. Also observe that, since the latter expression evaluates to something, so must $(insert\ l\ x)$. Let us call that value l'. Now suppose that $(find\ (Node\ (i, l, r)\ y)$ evaluates to true. This could happen in three ways.

- *Because y is i.* In this case, $(find\ (Node\ (i, insert\ l\ x, r))\ y)$ must obviously evaluate to true. But this expression is the same as $(find\ t'\ y)$, i.e. we have the desired conclusion.

- *Because $(y < i)$ and $(find\ l\ y)$ evaluates to true.* In this case, the induction hypothesis $P(l)$ easily yields the following: If $(find\ l\ y)$ evaluates to true then $(find\ l'\ y)$ evaluates to true). Thus, it follows that $(find\ l'\ y)$ evaluate to true. But then, using the definition of $find$, it follows that $(find\ (Node\ (i, l', r))\ y)$, i.e. $(find\ t'\ y)$ must also evaluate to true.

- *Because y is not less than i and $(find\ r\ y)$ evaluates to true.* In this case, using the definition of $find$, it follows easily that $(find\ (Node\ (i, l', r))\ y)$, i.e. $(find\ t'\ y)$, must evaluate to true.

Subcase x is not less than i. In this case, t' is whatever $Node\ (i, l, insert\ r\ x)$ evaluates to. Also, since the latter expression is assumed to evaluate to something, this must be true also for $(insert\ r\ x)$. Let us call that value r'. Now suppose that $(find\ t\ y)$ evaluates to true. This could happen in three ways.

- *Because y is i.* In this case, $(find\ (Node\ (i, l, insert\ r\ x))\ y)$ must obviously evaluate to true. But this expression is the same as $(find\ t'\ y)$, i.e. we have the desired conclusion.

- *Because $(y < i)$ and $(find\ l\ y)$ evaluates to true.* In this case, using the definition of $find$, it follows easily that $(find\ (Node\ (i, l, r'))\ y)$, i.e. $(find\ t'\ y)$, must evaluate to true.

- *Because y is not less than i and $(find\ r\ y)$ evaluates to true.* In this case, the induction $P(r)$ easily yields the following: If $(find\ r\ y)$ evaluates to true then $(find\ r'\ y)$ evaluates to true. But then, using the definition of $find$, it follows that $(find\ (Node\ (i, l, r'))\ y)$, i.e. $(find\ t'\ y)$ must also evaluate to true.

□

A curious thing about the proof above: it did not actually make use of the fact that a binary search tree respects an ordering. Try to understand why this is irrelevant to the fact that *insert* preserves all the elements in the original tree as probed by $find$. If this remains unclear, ask questions on Forum. You will want to understand a similar fact about the proof you do related to binary trees in Homework 5.