Lectures Notes on Proof by Induction

Construction and Verification of Software

Nova School of Science and Technology
Mário Pereira mjp.pereira@fct.unl.pt

Version of September 15, 2025

The contents of this document is strongly inspired by notes by Michael Erdmann and Frank Pfenning, and Bernardo Toninho.

1 Proof by Induction

The simplest form of induction, which you may recall from math or calculus classes, is induction over the naturals $0, 1, \ldots$. This is known by a few names: *mathematical induction*, *standard induction*, *simple induction* or *weak induction*. The idea is simple but powerful: assume we want to prove a property for every natural number n. We first prove that the property holds for the first natural number, 0 (the base case). Then, we assume the property holds for an arbitrary natural n and establish it for the successor of n (i.e. $n+1$ – the inductive step). The two steps together guarantee the property holds for all natural numbers – why? We know it holds for 0, and because of the inductive step, it must hold for its successor 1. By the same account, if it holds for 1 then it must hold for its successor 2, and so on, for any given natural number.

There are many small variations of this general scheme which can be easily justified and which we will also call mathematical induction. For instance, induction might start at 1 if we want to prove a property of all positive integers rather than the naturals.

As you may have guessed, proofs by induction are the main workhorse when reasoning formally about (recursive) functional programs. To see how it all works, consider the following recursive function:

```
let rec power (n: nat) (m: nat) : nat =                    OCaml
  match m with
  | O -> S O
  | S m' -> mult n (power n m')
```

The function above (inefficiently) computes n^m for an arbitrary values n and m by multiplying n with itself m times. We take $0^0 = 1$.

Lets now prove that `power n m` does indeed calculate n^m, using induction over the naturals.

Theorem 1.1. `power n m` $\Downarrow n^m$ *for all natural numbers* m *and* n.

Proof. By mathematical induction on $\mathtt{m}$.

- Base Case [$\mathtt{m} = \mathtt{0}$]:

  We need to show that `power n 0` $\Downarrow \mathtt{n}^{\mathtt{0}}$, for all $\mathtt{n}$.

	`power n 0`	
$\Longrightarrow^*$	`1`	(by simplification)

- Induction Step:

  Assume that, for some $\mathtt{m} \geq \mathtt{0}$, and all integers $\mathtt{n}$, `power n m` $\Downarrow \mathtt{n}^{\mathtt{m}}$. Prove that the property holds for $\mathtt{m}+\mathtt{1}$.

  We need to show that `power n (m + 1)` $\Downarrow \mathtt{n}^{\mathtt{m}+\mathtt{1}}$.

	`power n (m + 1)`	
$\Longrightarrow^*$	`n*power n (m + 1 - 1)`	(by simplification)
$\Longrightarrow^*$	`n*power n m`	(by math)
$\Longrightarrow^*$	$\mathtt{n}*\mathtt{n}^{\mathtt{m}}$	(by the induction hypothesis)
$\Longrightarrow$	$\mathtt{n} \times \mathtt{n}^{\mathtt{m}}$	(by evaluation of $*$)
$=$	$\mathtt{n}^{\mathtt{m}+\mathtt{1}}$	(by math)

  which completes the proof.

□

The level of detail of a proof typically depends on the context in which the proof is carried out (and the mathematical sophistication of the expected reader). In this course, you should feel free to omit the number of computation steps and combine obvious reasoning steps. Appeals to the induction hypothesis and other non-obvious steps (like appealing to lemmas) should be justified as in the example above.

You should convince yourself that the reasoning above does indeed prove the power function is correct: we show that `power n 0` is correct, for all $\mathtt{n}$ (base case); we show that, for some $\mathtt{m}$ and all $\mathtt{n}$, if we assume `power n m` is correct (the inductive hypothesis) then `power n (m + 1)` is correct (inductive case). Then, we can carry out this reasoning for an arbitrary $\mathtt{m}$: the function is correct for $\mathtt{0}$; by the inductive step, it is correct for $\mathtt{1}$; by the inductive step, it is correct for $\mathtt{2}$, and so on up-to $\mathtt{m}$. This is often the intuitive reasoning one performs when writing recursive functions.

2 Generalizing the Induction Hypothesis

From the example above it may seem that proofs by induction are always straightforward. While this is the case for many proofs, there is the occasional function whose correctness proof turns out to be more difficult. This is often because the statement we are trying to prove is *too weak*, and what we have to prove is something more general than the final result we are aiming for. This is referred to as *generalizing the induction hypothesis*. A common symptom of the need to generalize the induction hypothesis (i.h.) is when an appeal to the i.h. results in a statement that is overly specialized and does not actually allow us to proceed with our reasoning.

While it can be shown that there is no general recipe (`i.e.`, an algorithm) for this that will always work, we can isolate a common case. Consider the

following alternative implementation of the power function, through the use of a helper function:

```
let rec pow (n: nat) (m: nat) (r: nat) : nat =                    OCaml
  if eqb m 0 then r else pow n (m - 1) (r * n)

let power_alt (n: nat) (m: nat) : nat =
  pow n m 1
```

The `pow` function uses its third argument `r` as an *accumulator*, that is then used as the final result for `pow` when m is `0`. You should convince yourself that the definition of `power_alt` above and the definition of `power` from before are indeed equivalent.

As an aside, this is a commonly used trick in functional programming. In the earlier `power` function, if we want to calculate `power 2 4` we must calculate `power 2 3` and multiply it by `2`, which in turn requires calculating `power 2 2` and multiplying that intermediate result by `2`, and so on. Compilers for functional languages must make arrangements for storing the intermediate computations, so that the final result can be correctly "unrolled" back. On the other hand, the `pow` function above does not have this kind of recursive structure – it features so-called *tail recursion*. For instance, to calculate `pow 2 4 1` we need only calculate `pow 2 3 2`, and so `pow 2 2 4`, and so `pow 2 1 8` and finally `pow 2 0 16`, which is `16`. No "unrolling" back of intermediate results is needed. For functions of this shape, compilers can optimize away any special arrangements for storing intermediate computations, which can make the execution of tail-recursive functions such as `pow` much more efficient when compared to the execution of functions like `power`. This is called *tail-call optimization*.

We would now like to prove that `power_alt` n $m \Downarrow n^m$, for all naturals n and m. This requires proving something about `pow`, which takes one more argument. Intuitively, we would like to prove `pow` n m `1` $\Downarrow n^m$. Unfortunately, proving this statement directly by induction will fail. Lets see where it breaks down.

Suppose we have assumed the induction hypothesis:

$$\texttt{pow}\ n\ m\ \texttt{1} \Downarrow n^m$$

and try to prove

$$\texttt{pow}\ n\ (m+\texttt{1})\ \texttt{1} \Downarrow n^{m+1}$$

We proceed as we did in the inductive case for `power`:

$$\begin{array}{ll} & \texttt{pow}\ n\ m+\texttt{1}\ \texttt{1} \\ \Longrightarrow^* & \texttt{pow}\ n\ (m+\texttt{1}-\texttt{1})\ (\texttt{1}*n) \\ \Longrightarrow & \texttt{pow}\ n\ m\ (\texttt{1}*n) \\ \Longrightarrow & \texttt{pow}\ n\ m\ n \end{array}$$

but now we are stuck. We cannot apply the induction hypothesis since the last argument in the call to `pow` is not `1`. The solution is to generalize the property to allow any `r: nat` in such a way that the desired result follows easily. The generalization is as follows:

Lemma 2.1. `pow` n m $r \Downarrow r \times n^m$, *for all* r, *all* $n \geq 0$, *and all* m *natural.*

Proof. By mathematical induction on m.

Base Case: $m = 0$

We need to show that `pow n 0 r` $\Downarrow r \times n^0$, for all n.

	`pow n 0 r`	
$\Longrightarrow^*$	r	(by simplification)
$=$	$r \times n^0$	(by math)

Induction Step: Assume that, for some $m \geq 0$, and all integers n and r', `pow n m r'` $\Downarrow r' \times n^m$. Prove that the property holds for $m + 1$.

We need to show that `pow n (m + 1) r` $\Downarrow r \times n^{m+1}$.

	`pow(n,m + 1,r)`	
$\Longrightarrow^*$	`pow n (m + 1 − 1) (r*n)`	(by simplification)
$\Longrightarrow^*$	`pow n m (r × n)`	(by math and evaluation of $*$)
$\Longrightarrow^*$	$(r \times n) \times n^m$	(by the induction hypothesis, with $r' = r \times n$)
$=$	$r \times n^{m+1}$	(by math)

which completes the proof.

□

The correctness of `power_alt` can now be established directly.

Lemma 2.2. `power_alt n m` $\Downarrow n^m$*, for all natural numbers* n *and* m.

Proof.

	`power_alt n m`	
$\Longrightarrow$	`pow n m 1`	(by simplification)
$\Longrightarrow^*$	$1 \times n^m$	(by Lemma 2.1)
$=$	n^m	(by math)

□

3 Beyond Natural Numbers

Up to this point we have seen how to reason precisely about functions that manipulate and produce (natural) numbers through the proof method of mathematical induction. However, as you may recall from any course you may have participated in that featured functional programming, most interesting or realistic recursive functions manipulate more sophisticated data structures such as lists and trees – themselves often defined as recursive (or inductive) data types.

Consider for instance the following piece of code which defines a inductive data type `list` of natural numbers, and a recursive function that adds all the elements of a list:

```
type list = Nil | Cons of nat * list                    OCaml

let rec add (l : list) : nat =
  match l with
  | Nil -> 0
  | Cons x xs -> plus x (add xs)
```

The type has exactly two constructors, `Nil` and `Cons`. The `Nil` constructor codifies the empty list. The `Cons` constructor represents a (singly-linked) cell containing an element of type `nat` and (a reference to) the rest of the list.

The `add` function takes an argument `l` of type `list` and produces a value of type `nat` which is the sum of all elements of `l`. It does so by pattern matching on the structure of the given list. Since `l` is of type `list` it can only be constructed using a `Nil` or a `Cons`. In the former case, it means we are trying to add the elements of the *empty* list and so the result must be `O`. In the latter case, we are dealing with a list with at least one element and a (possibly empty) tail. Pattern matching allows us to *name* the components of the `Cons` cell so that we may use them directly, thus we write `| Cons x xs -> plus x (add xs)` to denote that when the list is a `Cons`-cell carrying a natural value `x` and rest of the list `xs`, the `add` function is defined by adding `x` to the result of `add`ing the rest of the list.

While it may be slightly harder to wrap your head around why, informally, the function `add` above is correct, intuitively we can see that it must produce the correct result: for empty lists it will produce `O`, for one element lists it will add that element to `O`, and so on. Moreover, since each call of `add` deconstructs one element of the list, and there can only be finitely many such elements, we can also convince ourselves that the function always terminates.

3.1 Structural Induction

We will now make the correctness argument precise (*i.e.*, formal) using a generalization of the technique from the previous sections: *structural induction*. The idea is that we can reason using induction on the possible *structure* of so-called inductive data types such as lists and trees in the sense given above. Instead of just postulating the induction principle over lists, we will attempt to construct it by analogy with the natural numbers.

Let us revisit the induction principle for natural numbers: to show that a property holds for all natural numbers we first prove that it holds for zero (*i.e.*, prove for `O`) and then, we assume the property holds for an arbitrary natural $\mathtt{n}'$ and prove the property holds for its sucessor (*i.e.*, assume property for $\mathtt{n}'$, prove for `S` $\mathtt{n}'$). The reason why this is a valid proof principle is the same as that for mathematical induction: values of type `nat` can only be either `O` or finitely generated by repeated application of `S` to a `nat`. Thus, for any given `nat`, if its `O`, the "base case" shows us the property holds; if its the successor of `O` (`S O`, the inductive step together with the base case shows us the property holds; if its `S S O`, the base case and two instances of the inductive step discharge the proof, and so on.

We can now reconstruct the induction principle for lists by following a similar recipe. We have as many cases as there are constructors in the data type: a base (or non-inductive case) for `Nil` and an inductive case for `Cons`. Spelling it out, to show a property holds for all values of type `list` we show the property holds for `Nil` and then, assuming it holds for an arbitrary list `l'`, we show that it holds for `Cons n l'`, for an arbitrary `n`. Let us show the `add` function over lists of integers is correct in the following sense:

Theorem 3.1. *For all* `l : list`, `add l` $\Downarrow \mathtt{l}_0 + \cdots + \mathtt{l}_n$, *where* $\mathtt{l}_i$ *denotes the* `i`*-th element of the list, and the length of* `l` *given by* `n`.

Proof. By structural induction on `l`.

Case: `l` is `Nil`

We need to show that `add(Nil)` $\Downarrow$ `0`

	`add Nil`	
$\Longrightarrow^*$	`0`	(by simplification)

Case: `l` is `Cons n l'`, for arbitrary `n` and `l'`

Assume that, `add l'` $\Downarrow \mathtt{l}_0 + \ldots \mathtt{l}_m$ where the length of `l'` is $\mathtt{m}$ (*i.e*, the induction hypothesis).

We need to show that `add (Cons n l')` $\Downarrow \mathtt{n} + \mathtt{l}_0 + \ldots \mathtt{l}_m$.

	`add (Cons n l')`	
$\Longrightarrow^*$	`n + add l'`	(by simplification)
$\Longrightarrow^*$	$\mathtt{n} + (\mathtt{l}_0 + \ldots \mathtt{l}_m)$	(by the induction hypothesis)
$=$	$\mathtt{n} + \mathtt{l}_0 + \ldots \mathtt{l}_m$	(by math)

which completes the proof.

□

3.2 Induction on Trees

Structural induction is by no means restricted to lists. In fact, one can reason by induction over any *recursively* defined data type. As mentioned before, when proceeding by induction, we have as many cases in our proofs as there are constructors in your type. Moreover, we have an induction hypothesis for recursive sub-component of a value.

Let us consider binary trees as our working ground. Such a data structure is defined in OCaml as follows:

```
type tree = Leaf | Node of tree * nat * tree                    OCaml
```

Now, we can define a simple function to manipulate values of type `tree`:

```
let rec mirror (t: tree) : tree =                                OCaml
  match t with
  | Leaf -> Leaf
  | Node (l, v, r) -> Node (mirror r, v, mirror l)
```

The above function simply takes the elements of a tree and, recursively, exchange them from left to right (and vice-versa), acting exactly as a mirror.

Now, time to prove properties about `mirror`. Consider the following Lemma:

Lemma 3.2 (MIRROR_INVOLUTIVE)**.** . $\forall$ `t : tree`, *`mirror (mirror t)`* $=$ `t`.

This statement tells the `mirror` function is an *involutive* operator. In practice, when called twice, it acts as the identity function. Lets prove the lemma.

Proof. By structural induction on `t`.

- Base case [`t = Leaf`]: trivial (Homework: why?).
- Inductive step [`t = Node (l, v, r)`]:

– In this case, we have two induction hypotheses, one for each sub-tree:

$$\texttt{mirror (mirror l) = l} \qquad (\dagger)$$

$$\texttt{mirror (mirror r) = r} \qquad (\ddagger)$$

– We proceed the proof as follows:

$$\begin{array}{lll}
 & \texttt{mirror (mirror t)} = \texttt{t} & \\
\Longrightarrow^* & \texttt{Node (mirror (mirror l), v, mirror (mirror r))} = \texttt{t} & \text{(by simplification)} \\
\Longrightarrow & \texttt{Node (l, v, mirror (mirror r))} = \texttt{t} & \text{(by rewriting with IH } \dagger\text{)} \\
\Longrightarrow & \texttt{Node (l, v, r)} = \texttt{t} & \text{(by rewriting with IH } \ddagger\text{)}
\end{array}$$

which completes the proof.

□