Induction

Brigitte Pientka*

January 19, 2007

In this note, we will briefly discuss how to prove properties about ML programs using induction. In particular, we will discuss proofs by "mathematical induction" and by "structural induction.

Proofs by induction are uqiquitous in the theory of programming languages, as in most of computer science. Many of these proofs are based on one of the following principles.

1 Mathematical induction

Mathematical induction is the simplest form of induction. When we try to prove a property for every natural number n, we first show the property holds for 0 (induction basis). Then we assume the property holds for n and establish it for $n + 1$ (induction step). Basis and step together ensure that the property holds for all natural numbers.

There are small variations of this scheme which can be easily justified. For example, we may start by proving the property holds for 1, if we want to prove a property for all positive integers. There may also be two base cases, one for 0 and one for 1.

As an example, let us consider the following program `power`.

```
(* Invariant: power:int >=0 *)
fun power(n, k)=
  if k=0 then 0  else n * power(n, k-1)
```

How can we prove that this program indeed computes n^k? – To clearly distinguish between the natural number n in our on-paper formulation and its representation and use in a program, we will write $\overline{n}$ to denote the latter. Moreover, we will use the following notation

$e \Downarrow v$	expression e evaluates in multiple steps to the value v.
$e \Rightarrow e'$	expression e evaluates in one steps to expresion e'.
$e \Rightarrow^* e'$	expression e evaluates in multiple steps to expresion e'.

In this example, we want to prove that `power(`$\overline{n}, \overline{k}$`)` evaluates in multiple steps to the value $\overline{n^k}$.

*These notes are partially inspired by some notes from F. Pfenning used at CMU.

Theorem 1.1. $\texttt{power}(\overline{n}, \overline{k}) \Downarrow \overline{n^k}$ for all $k \geq 0$

Proof. By induction on k.

Base Case $k = 0$

$$
\begin{array}{ll}
 & \texttt{power}(\overline{n}, \overline{0}) \\
\Rightarrow & \texttt{ if 0 = 0 then 0 else } \overline{n} \texttt{ * power(}\overline{n}\texttt{, 0-1)} \\
\Rightarrow & \texttt{ if true then 0 else } \overline{n} \texttt{ *power(}\overline{n}\texttt{,0-1)} \\
\Rightarrow & \texttt{0} = \overline{0} = \overline{n^0}
\end{array}
$$

Step Case Assume that $\texttt{power(}\overline{n}\texttt{,}\overline{k}\texttt{)} \Downarrow \overline{n^k}$. We have to show that $\texttt{power}(\overline{n}, \overline{k+1}) \Downarrow \overline{n^{k+1}}$.

$$
\begin{array}{lll}
 & \texttt{power(}\overline{n}\texttt{,} \overline{k+1}\texttt{)} & \\
\Rightarrow & \texttt{ if } \overline{k+1} \texttt{ = 0 then 0 else } \overline{n}\texttt{*power(} \overline{n}\texttt{,} \overline{k+1}\texttt{-1)} & \text{by program} \\
\Rightarrow & \texttt{if false then 0 else } \overline{n}\texttt{*power(}\overline{n}\texttt{,} \overline{k+1} \texttt{ - 1)} & \text{by program} \\
\Rightarrow & \overline{n} \texttt{ *power(}\overline{n}\texttt{,} \overline{k+1} \texttt{ - 1)} & \text{by program} \\
\Rightarrow & \overline{n} \texttt{ *power(}\overline{n}\texttt{,} \overline{k}\texttt{)} & \text{by program} \\
\Rightarrow & \overline{n} * \overline{n^k} & \text{by induction hypothesis} \\
\Rightarrow & \overline{n * n^k} & \text{by program} \\
\Rightarrow & \overline{n^{k+1}} & \text{by basic arithmetic}
\end{array}
$$

□

This proof emphasizes each step in the evaluation of the program. Often, we may not want to go trough each single step in that much detail. However, it illustrates that when reasoning about programs, we must know about the underlying operational semantics of the programming language we are using, i.e. how will a given program be executed.

2 Complete induction

The principle of complete induction formalizes a frequent pattern of reasoning. To prove a property by complete induction we first need to establish the induction basis for $n = 0$. Then we prove the induction step for $n \geq 0$ by assuming the property for all $n' < n$ and establishing it for n. One can think of it like mathematical induction, except that we are allowed to appeal to the induction hypothesis for any $n' < n$ and not just the immediate predecessor.

These two principles should be familiar to you and are the basis for proving some fundamental properties about programs. As an example, we define a program for `power` which is more efficient and defined via pattern matching.

```
fun power(n,0) = 1
  | power(n,k) =
   if even(k) then square(power(n, k div 2))
   else n * power (n, k-1)
```

To prove that this program works correctly, we rely on the following properties which we state as lemmas without proofs.

Lemma 2.1.
For all n, `square`$(\overline{n}) \Downarrow \overline{n^2}$.

Theorem 2.1. `power`$(\overline{(n)}, \overline{(k)}) \Downarrow \overline{n^k}$ for $k \geq 0$.

Proof. By complete induction on k.

Base Case $k = 0$

`power`$(\overline{n}, \overline{0})$
$\Rightarrow 1$

Step Case $k > 0$

Assume that `power`$(\overline{n}, \overline{k'}) \Downarrow \overline{n^{k'}}$ for any $k' < k$.
We have to show that `power`$(\overline{n}, \overline{k}) \Downarrow \overline{n^k}$.

`power`$(\overline{n}, \overline{k})$
$\Rightarrow$ `if even` $\overline{k}$ `then square(power(`$\overline{n}, \overline{k}$ `div 2))`
`else` $\overline{n}$ `* power (`$\overline{n}, \overline{k}$`-1)`

Now we will distinguish subcase, whether k is even or odd.

Sub-Case 1 $k = 2k'$ for some $k' < k$.

$\Rightarrow$ `square(power(`$\overline{n}, \overline{2k'}$`div 2))`
$\Rightarrow$ `square(power(`$\overline{n}, \overline{k'}$`))`
$\Rightarrow$ `square(`$\overline{n^{k'}}$`)` by i.h. on k'
$\Rightarrow$ $\overline{(n^{k'})^2}$ by Lemma 1
$=$ $\overline{n^{2k'}} = \overline{n^k}$

Sub-Case 2 $k = 2k' + 1$ for some $k' < k$.

$\Rightarrow$ $\overline{n}$ `* power (`$\overline{n}, \overline{k}$`-1)`
$\Rightarrow$ $\overline{n}$ `* power (`$\overline{n}, \overline{k-1}$`)`
$\Rightarrow$ $\overline{n}$ `*` $\overline{n^{k-1}}$ by i.h. $k-1$
$\Rightarrow$ $\overline{n * n^{k-1}} = \overline{n^k}$

□

3 Structural induction

When proving properties about ML programs, we typically need to reason not only about numbers but about defined inductively data-structures, such as lists, trees etc. Structural induction allows us to reason about the structure of the objects we are considering. This is best illustrated by considering an example of an inductive data-type such as lists.

```
datatype 'a list = nil | :: of 'a * 'a list
```

To inductively prove a property about lists, we first prove it for the empty list `nil`. Then we assume the property holds for lists `t` and establish it for lists `h::t`.

Similarly, for trees:

```
datatype 'a tree = Empty | Node of 'a * 'a tree * 'a tree
```

To inductively prove a property about trees, we first prove it for the emtpy tree `Empty`. Then we assume the property holds for trees `L` and `R`, and establish it for the tree `Node(a, L, R)`.

Inductive data-structures make it easy to reason about them inductively, since they directly give rise to induction principles. Typically, we reason directly about their structure. For example, we consider `L` and `R` to be sub-trees of the tree `Node(a, L, R)`. Let us consider the following two programs. The first one allows us to insert an element, which consists of a key `x` and the data `d`, into a binary search tree. The second one allows us to lookup the data `d` associated with some key `x` in a binary search tree `T`.

```
fun insert (e as (x,d)) Empty = Node(e, Empty, Empty)
 | insert (e as (x,d)) (Node((y,d'), L, R)) =
 if x = y
   then Node(e, L, R)
 else
   (if x < y then
      Node((y,d'), insert e L, R)
    else
      Node((y,d'), L, insert e R))

fun lookup x Empty = NONE
  | lookup x (Node((y,d), L, R)) =
  if x = y then SOME(d)
    else
      (if x < y then lookup x L
         else lookup x R)
```

Let's try to prove that when we have inserted an element `(x,d)` into a binary search tree `T`, and then look up the data corresponding to the key `x`, we will get back the date `d`. In the proof below we will write $\Rightarrow *$ when we skip over some intermediate steps.

Theorem 3.1. If `T` is a binary search tree, then `lookup x (insert (x,d) T)` $\Downarrow$ `SOME(d)`

Proof. By structural induction on `T`.

Base Case `T = Empty`

	`lookup x (insert (x,d) Empty)`	
$\Rightarrow$	`lookup x (Node((x,d), Empty , Empty)`	by program insert
$\Rightarrow$	`SOME(d)`	by program lookup

Step Case `T = Node((y,d'), L, R)`

We can assume the property holds for the sub-trees `L` and `R`.

1. `lookup x (insert (x,d) L` $\Downarrow$ `SOME(d)`

2. `lookup x (insert (x,d) R` $\Downarrow$ `SOME(d)`

Sub-case: `x = y`

	`lookup x (insert (x,d) (Node((y,d'), L, R)))`	
$\Rightarrow^*$	`lookup x (Node((x,d), L, R))`	by program insert
$\Rightarrow^*$	`SOME(d)`	by program lookup

Sub-case: `x < y`

	`lookup x (insert (x,d) (Node((y,d'), L, R)))`	
$\Rightarrow^*$	`lookup x (Node((y,d'), insert (x,d) L, R))`	by program insert
$\Rightarrow^*$	`lookup x (insert (x,d), L)`	by program lookup
$\Rightarrow$	`SOME(d)`	by i.h.

Sub-case: `y < x`

	`lookup x (insert (x,d) (Node((y,d'), L, R)))`	
$\Rightarrow^*$	`lookup x (Node((y,d'), L, insert (x,d) R))`	by program insert
$\Rightarrow^*$	`lookup x (insert (x,d), R)`	by program lookup
$\Rightarrow$	`SOME(d)`	by i.h.

□

4 Generalizing the statement

From the examples, it may seem that induction is always straightforward. Often this is indeed the case. Sometimes however we will encounter functions whose correctness property is more difficult to prove. This is often because we need to prove something more general than the final result we are aiming for. This is also referred to as *generalizing the induction hypothesis*. There is no general recipe for generalzing the induction hypothesis, but one common case is the following.

Consider the following two programs for reversing a list. The first one is the naive version, while the second one is the tail-recursive version.

```
fun rev([]) = []                fun rev' ([], acc) = acc
  | rev(x::t) = rev(t)@[x]        | rev' (x::t, acc) = rev'(t, x::acc)
```

We would like to prove that both programs yield the same result. Essentially we would like to say `rev(l)` returns the same result as calling `rev'(l,[])`.

$$\texttt{rev(l)} \Downarrow \texttt{l'} \text{ iff } \texttt{rev'(l,[])} \Downarrow \texttt{l'}$$

We will simplify this statement a little bit, and try to prove

$$\texttt{rev(l)} = \texttt{rev'(l,[])}$$

We will use this as an abbreviation for the more verbose statement, if `rev(l)` $\Downarrow$ `r` and `rev'(l,[])` $\Downarrow$ `r'` then `r` = `r'`.

The problem arises in the step-case, when we attempt to prove

$$\texttt{rev(x::t)} = \texttt{rev'(x::t, [])}$$

On the left, the program `rev'` evaluates as follows:

```
     rev'(x::t, [])
 ⇒   rev'(t, x::[])
 ⇒   rev'(t, [x])
```

But now we are stuck. We cannot apply the induction hypothesis, because the statement we attempt to prove requires that the second argument to `rev'` is the empty list! The solution is to generalize the statement in such a way that the desired result follows easily.

The following theorem generalizes the problem appropriately.

Theorem 4.1. For any list `l`, `rev(l)@acc` = `rev'(l, acc)`

Proof. Induction on `l`.

Base Case `l = []`

```
     rev([])@acc
 ⇒   []@acc                    by program rev
 ⇒   acc                       by program @
 ⇐   rev'([], acc)             by program rev'
```

Step Case `l = x::t`

Assuming, for any `acc'`, `rev(t)@acc'` = `rev'(t,acc')`,
we must prove `rev(x::t)@acc'` = `rev'(x::t, acc')` .

	`rev(x::t)@acc`	
$\Rightarrow$	`(rev(t)@[x])@acc`	by program `rev`
$\Rightarrow$	`rev(t)@([x]@acc)`	by associativity of `@`
$\Rightarrow$	`rev(t)@(x::acc)`	by Lemma
$=$	`rev'(t, x::acc)`	by i.h.
$\Leftarrow$	`rev'(x::t, acc)`	by program

□

We want to emphasize that you should always state lemmas (i.e. properties) you are relying on. In the previous example, we need to know properties of append for example.

5 Conclusion

We presented several important induction principles and examples of induction proofs. While in practice, we will rarely verify programs completely, we may want to prove certain properties about them in practice, for example we may want to prove that some confidential data is not leaked, or only some designated principals will have access to a given resource. These properties will typically follow the same induction principles we have seen in these notes.

There is a wide spectrum of properties we would like to enforce about programs. Types, as we encounter them in a language such as SML, enforce fairly simple properties. For example, the type of the lookup function is `int * (int * 'a) tree -> 'a option`. While this gives us a partial correctness guarantee, it does for example not ensure that the tree passed is a binary search tree. On the other hand, type systems are great because they enforce a property statically. When you change your program, the type-checker will verify if it still observes this type property. If it doesn't the type checker will give precise error messages, so the programmer can fix the problem. Inductive proofs can typically enforce stronger properties about programs than types. In fact, we can prove full correctness. However, inductive proofs have to be redone every single time your program changes. Doing them by hand is time-consuming. What is it we actually need to prove? How do we know when to generalize an induction hypothesis? What happens if a proof fails? Can we give meaningful error messages in this case? A key question is therefore how we can make type systems stronger so they can check stronger properties statically, while retaining all their good properties. But that's a question for a different course :–).