CSci 2041: Advanced Programming Principles

Spring 2018

Eric Van Wyk

Sample proofs by induction

This document contains sample inductive proofs of small recursive functions, typically operating on recursive data. Many of these are ones that we discussed in lecture.

The sumTo function over natural numbers Consider the following function:

```
let rec sumTo n = match n with
  | 0 -> 0
  | n -> n + sumTo (n-1)
```

We would like to show that

$$\forall n : \texttt{sumTo n} = 0 + 1 + 2 + ... + n$$

The principle of induction for natural numbers is:

$$\forall n, P(n) \text{ if } P(0) \text{ and } P(n) \Rightarrow P(n+1)$$

In this case, our property P is defined as

$$P(n) \text{ is } \texttt{sumTo n} = 0 + 1 + 2 + ... + n$$

Our inductive proof would have two cases:

- P(0): show that `sumTo 0` $= 0$

  This is quite simple.

  `sumTo 0`

  $=$ 0, by definition of sumTo

- P(n+1): show that `sumTo (n+1)` $= 0 + 1 + 2 + ... + (n+1)$ where `sumTo n` $= 0 + 1 + ... + n$ holds by the inductive hypothesis.

  `sumTo (n+1)`

  $=$ (n+1) + `sumTo ((n+1)-1)`, by definition of sumTo

  $=$ (n+1) + `sumTo n`, by properties of addition and substraction

  $=$ (n+1) + $0 + 1 + ... + n$, by inductive hypothesis

  $= 0 + 1 + ... + n + (n+1)$, by properties of addition

  $= 0 + 1 + ... + (n+1)$, by simplifying the "$0 + 1 + ... + x$ notation.

Note that at each step, we make a note to justify why one expression has the same value as another. This is the form that your proofs will have.

Functions over the `nat` structured type Consider the following OCaml types and functions

```
type nat = Zero | Succ of nat

let toInt = function
  | Zero -> 0
  | Succ n -> toInt n + 1

let rec add n1 n2 = match n1, n2 with
  | Zero, n2 -> n2
  | Succ n1', n2 -> Succ (add n1' n2)
```

The principle of induction for type `nat` is:

$$\forall n,\ P(n) \text{ if } P(\texttt{Zero}) \text{ and } P(n) \implies P(\texttt{Succ}\ n).$$

We would like to show that

$$\forall n, n' \in \texttt{ nat . toInt (add } n\ n'\texttt{)} = \texttt{toInt } n + \texttt{toInt } n'$$

Thus, our property $P(n)$ is

$$\forall n' \in \texttt{nat . toInt (add } n\ n'\texttt{)} = \texttt{toInt } n + \texttt{toInt } n'$$

Notice that now our property contains a universal quantifier ($\forall$).
Our proof proceeds by induction P (which is the first argument to `add`). We need to show

- P(Zero): $\forall n' \in$ `nat.toInt (add Zero n2) = toInt Zero + toInt n2`

  For this proof, our equational reasoning is taking place under (inside of) the quantifier for n'. This can be seen below in which each step in the reasoning is written inside this quantifier.

  $\forall n' \in \texttt{nat}.($

  $\quad\texttt{toInt (add Zero } n'\texttt{)}$
  $\quad= \texttt{toInt } n'$ by definition of `add`
  $\quad= 0 + \texttt{toInt } n'$, by identity of addition of natural numbers
  $\quad= \texttt{toInt Zero} + \texttt{toInt } n'$, by definition of `toInt`

)

- $P(\texttt{Succ } n)$: $\forall n' \in \texttt{nat . toInt (add (Succ } n\texttt{) } n'\texttt{)} = \texttt{toInt (Succ } n\texttt{)} + \texttt{toInt } n'$
  given: $\forall n' \in \texttt{nat . toInt (add } n\ n'\texttt{)} = \texttt{toInt } n + \texttt{toInt } n'$

  $\forall n' \in \texttt{nat}.($

  $\quad\texttt{toInt (add (Succ } n \texttt{) } n'\texttt{)}$
  $\quad= \texttt{toInt (Succ (add } n\ n'\texttt{)}$, by def. of `add`
  $\quad= \texttt{toInt (add } n\ n'\texttt{)} + 1$, by def. of `toInt`
  $\quad= \texttt{toInt } n + \texttt{toInt } n' + 1$, by induction hypothesis
  $\quad= \texttt{toInt } n + 1 + \texttt{toInt } n'$, by commutativity of addition
  $\quad= \texttt{toInt (Succ } n\texttt{)} + \texttt{toInt } n'$, by def of `toInt`

)

Lists Lists are inductive types and we can thus write straightforward inductive proofs for recursive functions over lists. Recall, the list type:

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

The principle of induction for this type is:

$\forall \ell, P(\ell)$ if $P($`[ ]`$)$ and
$\qquad P(\ell') \Longrightarrow P($`v ::` $\ell'))$

Consider the following function for adding up integer values in a list:

```
let rec sum = function
  | [] -> 0
  | x:xs -> x + sum xs
```

We might like to show that:

$$\forall \ell_1, \ell_2 \in \texttt{'a list . sum (} \ \ell_1 \texttt{ @ } \ell_2\texttt{) = sum } \ell_1 + \texttt{sum } \ell_2$$

Our property $P(\ell_1)$ is

$$\forall \ell_2 \in \texttt{'a list . sum (} \ \ell_1 \texttt{ @ } \ell_2\texttt{) = sum } \ell_1 + \texttt{sum } \ell_2$$

This will lead to a proof by induction over ℓ_1. This is the first argument to `sum`. We need to show:

- P(`[]`,r): `sum ([] @ r) = sum [] + sum r`

  $\forall r \in$ `'a list` . (

  $\qquad$ `sum ([] @` r`)`

  $\qquad = $ `sum` r, by properties of lists, namely that `[] @ l` $=$ `l`

  $\qquad = 0 +$ `sum` r, by properties of addition

  $\qquad = $ `sum [] + sum` r, by definition of sum

)

- P(`x::xs`, `r`): `sum (x:xs @ r) = sum (x:xs) + sum r`

  By the induction hypothesis we are given `sum (xs @ r) = sum xs + sum r`

  $\forall r \in$ `'a list` . (

  $\qquad$ `sum (x:xs @` r`)`

  $\qquad = $ `sum (x :: (xs @` r `))`, by properties of lists

  $\qquad = $ x $+$ `sum (xs @` r`)`, by def of `sum`

  $\qquad = $ x $+$ `sum xs + sum` r, by induction hypothesis

  $\qquad = $ `sum (x:xs) + sum` r, by def of `sum`

)

Ordered lists Consider the functions for ordered lists from the `ordered_list.ml` file in the code-examples directory of the public class repository:

```
let rec place e l = match l with
  | [ ] -> [e]
  | x::xs -> if e < x then e::x::xs else x :: (place e xs)

let rec is_elem e l = match l with
  | [ ] -> false
  | x::xs -> e = x || (e > x && is_elem e xs)
```

We might like to show that P(`e`,`l`): `is_elem e (place e l) = true` holds. Here we will write a property over two values, but we can separate out the value over which we are **not** doing induction with a quantifier as we've done above. But with a bit of care, we can write our proofs this way as well.
Our proof proceeds by induction over the list in P. We need to show:

- Base case: P(`[]`,`e`): `is_elem e (place e []) = true`.

  `is_elem e (place e [])`
  = `is_elem e [e]`, by definition of `place`
  = `e = e || e > e && is_elem e []`, by definition of `is_elem`
  = `true`, since `e` = `e`

- Inductive case: P(`y::ys, e`): `is_elem e (place e (y::ys)) = true`.

  By the inductive hypothesis we are given that `is_elem e (place e ys)` holds.

  We break the inductive case down into three sub-cases:

  Case: $e < y$

  `is_elem e (place e (y::ys))`
  = `is_elem e (e::y::ys)`, by definition of `place` and by assumption for this case
  = `e = e || ...`, by definition of `is_elem`
  = `true`, since `e` = `e`

  Case: $e = y$

  `is_elem e (place e (y::ys))`
  = `is_elem e (y ::  place e ys)`, by definition of `place`
  = `e = y || e > y && is_elem e (place e ys)`, by definition of `is_elem`
  = `true` by the assumption of this case that $e = y$.

  Case: $e > y$

  `is_elem e (place e (y::ys))`
  = `is_elem e (x ::  place e xs)`, by definition of `place`
  = `e = y || e > y && is_elem e (place e ys)`, by definition of `is_elem`
  = `e > y && is_elem e (place e ys)`, by def of is_elem
  = `is_elem e (place e ys)`, by assumption of this case that $e > y$
  = `true`, by the induction hypothesis

Proofs over recursive structure of the computation Recall our function `euclid` for computing the greatest common divisor of two positive integers.

```
let rec euclid m n =
  if m = n then m
  else
    if m < n
    then euclid m (n-m)
    else euclid (m-n) n
```

Our specifications for greatest common divisor were as follows:

$$\begin{array}{lcll} gcd\ m\ n & = & gcd\ m\ (n-m) & \text{if } n > m \\ gcd\ m\ n & = & gcd\ (m-n)\ n & \text{if } m > n \\ gcd\ m\ n & = & m & \text{if } m = n \end{array}$$

We did not use induction over natural numbers or some form of structured data in reasoning about the correctness of this function. Instead we used induction over the recursive nature of the computation itself.

We would like to show that:

$$P(m,n)\text{: } \texttt{euclid m n} = gcd\ m\ n$$

Our proof proceeds by induction over the recursive calls in `euclid`.

- Base case: show the property holds for non-recursive calls when $m = n$.

  `euclid m n`
  = `m`, by definition of `euclid` and assumption of the base case that $m = n$
  = $gcd\ m\ n$, by definition of gcd

- Inductive case: show it holds for recursive calls, assuming that the invariant holds on previous recursive calls.

  Case: $m < n$

  `euclid m n`
  = `euclid m (n-n)`, by definition of `euclid` and assumption of this case that $m < n$
  = $gcd\ m\ (n-m)$, by the inductive hypothesis
  = $gcd\ m\ n$, by definition of gcd

  Case: $m > n$

  `euclid m n`
  = `euclid (m-n) n`, by definition of `euclid` and assumption of this case that $m > n$
  = $gcd\ (m-n)\ n$, by the inductive hypothesis
  = $gcd\ m\ n$, by definition of gcd

For this proof by induction to work we need to know that the function will terminate.

This is not too difficult to see. The pair of value, `m`, `n`, can be seen as decreasing with each call. Here, "decreasing" is the lexicographic ordering on the pair `(m, n)`: `(m,n)` $<$ `(m',n')` if `m` $<$ `m'` or `m` $=$ `m'` and `n` $<$ `n'`.

Consider a few example calls to `euclid` if this notion of ordering is not clear to you.

This subtraction may make the value become equal, in which case the recursion terminates, or since `m` and `n` are initially positive and since any subtraction of one from the other is always a subtraction of the smaller from the strictly larger, the result will not be smaller than the initially smaller one and the resulting pair is lexicographically smaller.

Loop invariants and imperative programming In the functional implementation of `euclid` above the property P that we proved was `euclid m n` $=$ *gcd m n*. The key to this proof was that every call to `euclid` evaluated to *gcd m n*. The first, the second, etc. all the way to the last call which is the non-recursive base case. In imperative programs we do something similar with a loop invariant. There is some property that we wish to hold after every execution of the loop body. This invariant is maintained as each execution of the loop body makes some progress in the computation. After the loop has ended, the negation of the loop condition must be true (assuming there are no break statements in the loop body). The negation of the loop condition and the loop invariant should be enough to imply that the post condition should hold.

Recall our example from class:

```
(* Pre condidtion: m > 0, n > 0 *)
x = m
y = n

(* Loop invariant: gcd m n = gcd x y *)
while x <> y {
  if x > y
    x = x - y
  else
    y = y - x
}
(* Answer stored in x *)
(* Post condidtion: gcd m n = x *)
```

With the pre-condition, the loop invariant, and the post-condition we can argue, informally but rigorously, that the above code snippet is "partially correct." Partial correctness argues that *if* the code terminates, then it will have computed the correct value.

(An argument for "total correctness" would consist of an argument for partial correctness and an argument that the code does terminate.)

This argument consists of three components:

1. Show/argue that the precondition implies the loop invariant holds before entering the loop.
2. Show/argue that if it holds before the loop, it holds after a single iteration of the loop.
3. Show/argue that after the loop terminates, the negation of the condition and the loop invariant (which we know will be true) implies the post condition.

In our example `euclid` program these can be stated as follows:

1. Since `x` gets the value of `m` and `y` gets the value of `n`, clearly `gcd m n = gcd x y`. Thus the loop invariant holds before the loop.

2. To argue that the loop invariant is maintained after an iteration of the loop we assume that it holds before the iteration begins.

   Thus `gcd m n = gcd x y` is true before the loop body.

   There are two cases to consider:

   (a) Case `x > y`: For `gcd m n = gcd x y` to be true after the assignment `x = x - y` it should be the case that `gcd m n = gcd (x-y) y` holds before the assignment.

   Since `x > y`, and by the definition of *gcd*, `gcd m n = gcd (x-y) y` is equivalent to `gcd m n = gcd x y` which does hold at the beginning of the loop.

   Thus if `gcd m n = gcd x y` holds at the beginning of the loop body and `x > y` then `gcd m n = gcd x y` holds at the end.

   (b) Case `y > x`: Similarly, for `gcd m n = gcd x y` to be true after the assignment `y = y - x` it should be the case that `gcd m n = gcd x (y - x)` holds before the assignment.

   Since `y > x`, and by the definition of *gcd*, `gcd m n = gcd x (y-x)` is equivalent to `gcd m n = gcd x y` which does hold at the beginning of the loop.

   Thus if `gcd m n = gcd x y` holds at the beginning of the loop body and `y > x` then `gcd m n = gcd x y` holds at the end.

   Thus if `gcd m n = gcd x y` holds at the beginning of the loop body it holds at the end.

3. Since the loop has terminated `x = y` and thus `gcd x y = x`. Since the loop invariant `gcd m n = gcd x y` holds we know that `gcd m n = x` holds.