Advanced Programming Topics

Proof in the wild: strong induction, imperative proof, automated proof
Lecture 6
James Marshall

Strong Induction

Sometimes we need to consider more than just the next-smallest case in constructing a proof by induction.

e.g. the Fibonacci numbers (from lecture 1)

```
f 1 = 1                                                    --(fib1)
f 2 = 1                                                    --(fib2)
f n = f (n-1) + f (n-2)                                    --(fibn)
```

N.B. from lecture 1 we know that `f 100` takes longer than the universe has been in existence to compute. However, we should be able to prove that the right answer would be given! For example we should be able to prove `fibn`, since we know that the *n*-th Fibonacci number is given by the formula

Q: …*but*, how can we formulate an inductive hypothesis involving only the previous number in the sequence, when we need to know the previous *two*?

A: by having our inductive hypothesis be that the desired property holds for all $k < n$

Exercise (hard): use the mathematical definition above to prove `f n` *is correct for all* `n` ≥ 0

N.B. it may seem that strong induction is more powerful than weak induction, but actually

principle of weak induction principle of strong induction

Q: why?

Strong Induction Example - Integer Exponentiation

```
fastIntExp :: Integer -> Integer -> Integer
fastIntExp a b
    | b == 0    = 1                                          --(exp0)
    | even b    = (fastIntExp a (half b))^2                --(expeven)
    | otherwise = (fastIntExp a (b-1)) * a                  --(expodd)
        where half x = truncate ((fromIntegral x)/2)
```

Prove that for all defined value `fastIntExp a n == a^n`

1. Prove the base case:

2. Prove the induction step under the inductive hypothesis `fastIntExp a k == a^k` for `k<n`

Structural induction is implicitly strong…

Strong Structural Induction Example - Binary Tree Size (from Thompson chapter 14)

Prove that the number of nodes in a binary tree is strictly less than its 2 to the power of its depth.

```
data NTree = NilT |
             Node Int NTree NTree

depth NilT           = 0                                  --(depth.1)
depth (Node n t1 t2) = 1 + max (depth t1) (depth t2)   --(depth.2)

size NilT            = 0                                   --(size.1)
size (Node x t1 t2)  = 1 + size t1 + size t2               --(size.2)
```

1. State the proof goal as proving for any `NTree tr` that `size tr < 2^(depth tr)`

2. Prove the base case

3. Prove the induction step, that the property holds for any tree, given the induction hypothesis that the property holds for that tree's left and right subtrees

Proving Correctness in Imperative Programs

Most imperative languages do not use recursion, but instead use loops. But loops are very similar to recursion, so we can adapt proof by induction to prove the correctness of loops, by using *loop invariants.* These feature in imperative proof systems such as *Floyd-Hoare logic*, which reason about pre and post-conditions for the execution of functions. Note that fully automated proof is impossible however; if we could specify termination as a post-condition of a function and prove it automatically, we would have solved the *halting problem* (see COM2003 notes).

Definition: Loop Invariant

A *loop invariant* is a property of a loop that satisfies the following conditions (*e.g.* CLRS chapter 2):

1. *Initialisation*: The loop invariant is true before the first iteration of the loop.

2. *Maintenance:* If the invariant is true at the start of a loop iteration then it is also true at the end of the loop iteration.

3. *Termination:* When the loop terminates the invariant helps prove the correctness of the algorithm implemented by the loop.

Loop Invariant Example - Insertion Sort (from chapter 2 of CLRS)

```
INSERTION-SORT(A)
1       for j ⟵ 2 to length[A]
2               do key ⟵ A[j]
3                       Insert A[j] into the sorted sequence A[1 .. j − 1]
4                       i ⟵ j − 1
5                       while i > 0 and A[i] > key
6                               do A[i + 1] ⟵ A[i]
7                                       i ⟵ i − 1
8                       A[i + 1] ⟵ key
```

(Outer) Loop invariant:
At the start of each iteration the subarray $A[1 .. j-1]$ *is a sorted version of the original subarray* $A[1 .. j-1]$

Initialisation:
At initialisaation $A[1 .. j-1] = A[1 .. 1]$ *contains 1 element, so the loop invariant is trivially true*

Maintenance:
Informally, each loop iteration moves the new element to is correct position in the subarray. Formally, could use a loop invariant on the inner loop.

Termination:
When the loop ends $j = n + 1$*, where n is the length of the array to be sorted. By the loop invariant, at termination the array* $A[1 .. j-1] = A[1 .. n]$ *is a sorted version of the original* $A[1 .. n]$