12. Structural Induction

Language & Logic

Dave Parker

University of Birmingham

2017/18

Overview

- Proof by induction
  - mathematical induction
  - example
- Structural induction
  - recursive programs on recursive data structures
  - examples
- Exercise classes
  - Tue and Thu next week…

Proof by induction

- Mathematical induction
  - proof technique for statements of the form $\forall n\ [\ P(n)\ ]$
  - where n is a natural number

$$\frac{P(0) \qquad \forall k\ [\ P(k) \rightarrow P(k+1)\ ]}{\forall n\ [\ P(n)\ ]}$$

- Two steps:
  - 1. Base case:
    - e.g. prove that $P(0)$ is true
  - 2. Inductive step:
    - assume that $P(k)$ is true, prove that $P(k+1)$ is true
    - $P(k)$ is called the inductive hypothesis

- Conclude
  - $P(n)$ is true for all n

Mathematical induction – Example

- Prove: $\forall n\ [\ P(n)\ ]$ where $P(n)$ is $\Sigma_{i=0\ldots n}\ i = n(n+1)/2$

- 1. Base case (n=0):
  - LHS:
    $\Sigma_{i=0\ldots n}\ i = 0$
  - RHS:
    $n(n+1)/2 = (0\text{x}1)/2$
    $= 0$
  - so $P(0)$ is true

- 2. Inductive step
  - assume inductive hypothesis $P(k)$
    i.e., $\Sigma_{i=0\ldots k}\ i = k(k+1)/2$
  - then prove $P(k+1)$
    i.e., $\Sigma_{i=0\ldots k+1}\ i = (k+1)(k+2)/2$
  - $\Sigma_{i=0\ldots k+1}\ i = \ldots$

    $= (k+1)(k+2)/2$

Mathematical induction – Example

- Prove: $\forall n\ [\ P(n)\ ]$ where $P(n)$ is $\Sigma_{i=0\ldots n}\ i = n(n+1)/2$

- 1. Base case (n=0):
  - LHS:
    $\Sigma_{i=0\ldots n}\ i = 0$
  - RHS:
    $n(n+1)/2 = (0\times1)/2$
    $= 0$
  - so $P(0)$ is true

- 2. Inductive step
  - assume inductive hypothesis $P(k)$
    i.e., $\Sigma_{i=0\ldots k}\ i = k(k+1)/2$
  - then prove $P(k+1)$
    i.e., $\Sigma_{i=0\ldots k+1}\ i = (k+1)(k+2)/2$
  - $\Sigma_{i=0\ldots k+1}\ i = (0+1+\ldots+k+k+1)$
    $= (\Sigma_{i=0\ldots k}\ i) + (k+1)$
    $= k(k+1)/2 + (k+1)$
    (using inductive hypothesis)
    $= (k(k+1) + 2(k+1))/2$
    $= (k^2+3k+2)/2$
    $= (k+1)(k+2)/2$
  - so $P(k) \rightarrow P(k+1)$

- And therefore $\forall n\ [\ P(n)\ ]$

Next up: Structural induction

- Inference rule:

$$\frac{P(0) \quad \forall k\,[\,P(k) \rightarrow P(k+1)\,]}{\forall n\,[\,P(n)\,]}$$

- We could rewrite this as:

$$\frac{P(0) \quad \forall k\,[\,P(k) \rightarrow P(\mathrm{succ}(k))\,]}{\forall n\,[\,P(n)\,]}$$

- And in fact "succ" could be any recursive definition…

Structural induction

- Structural induction
  - proof method: used to prove properties of recursively defined objects or data structures (e.g. lists, trees)
  - prove that some property $P(x)$ holds for all instances x
  - $P(x)$ will usually relate to (recursively defined) functions
  - allows us to reason about correctness of programs

- Generalises mathematical induction
  - using notion of recursion
  - split into base case(s) & inductive step (recursive case)
  - and then infer that $\forall x\, [\, P(x)\,]$

Structural – Programs & lists

- For our examples, we'll use functional programs
  - often easier to prove/argue that these are correct
  - we'll work with examples written in Ocaml

- For now, we'll work with lists
  - defined recursively as a head and a tail
  - notation: hd::tl, where hd is a list item and tl is another list
  - the empty list is written []

Example 1

- Two OCaml functions operating on lists
(defined recursively, using pattern matching):
  - length l returns the length of list l
  - append l1 l2 returns the concatenation of lists l1 and l2

```
let rec length l = match l with
| [] -> 0
| hd::tl -> 1 + length tl
```

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

Example 1

- Prove:
  - the size of the concatenation of two lists is always equal to the sum of the sizes of the two individual lists
  - length (append l1 l2) = length l1 + length l2
    - (written in Ocaml notation)

```
let rec length l = match l with
| [] -> 0
| hd::tl -> 1 + length tl
```

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

Example 1

- Prove: length (append l1 l2) = length l1 + length l2

```
let rec length l = match l with
| [] -> 0
| hd::tl -> 1 + length tl
```

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

- 1. Base case (l1 = [])
  - LHS:
    - length (append [] l2) = length l2
  - RHS:
    - length [] + length l2 = 0 + length l2 = length l2

Example 1

- Prove: length (append l1 l2) = length l1 + length l2

```
let rec length l = match l with
| [] -> 0
| hd::tl -> 1 + length tl
```

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

- 2. Inductive step (l1 = hd::tl)
  - inductive hypothesis: length (append tl l2) = length tl + length l2
  - prove: length (append hd::tl l2) = length hd::tl + length l2
  - start with LHS:
    - length (append (hd::tl) l2)

= length (hd :: (append tl l2))	(using definition of append)
= 1 + length (append tl l2)	(using definition of length)
= 1 + length tl + length l2	(using inductive hypothesis)
= length hd::tl + length l2	(using definition of length)

Example 1

- Prove: length (append l1 l2) = length l1 + length l2

```
let rec length l = match l with
| [] -> 0
| hd::tl -> 1 + length tl
```

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

- 2. Inductive step (l1 = hd::tl)
  - inductive hypothesis: length (append tl l2) = length tl + length l2
  - prove: length (append hd::tl l2) = length hd::tl + length l2
  - LHS = RHS

- And therefore:
  - ∀l1 [length (append l1 l2) = length l1 + length l2]
  - and: ∀l2 [∀l1 [length (append l1 l2) = length l1 + length l2]]

Example 2

- Two more OCaml functions operating on lists (again, defined recursively, using pattern matching):
  - mem x tests for membership of item x in a list
  - occ x counts how many times item x occurs in a list

```
let rec mem x = function
| [] -> false
| hd::tl -> (hd = x) || (mem x tl)
```

```
let rec occ x = function
| [] -> 0
| hd::tl -> (if hd = x then 1 else 0) + occ x tl
```

Example 2

- Prove:
  - if mem says an element x is not in a list l,
    then occ returns 0 for the number of occurrences of x in l
  - if not (mem x l) then occ x l = 0
    - (again, written in Ocaml notation)

```
let rec mem x = function
| [] -> false
| hd::tl -> (hd = x) || (mem x tl)
```

```
let rec occ x = function
| [] -> 0
| hd::tl -> (if hd = x then 1 else 0) + occ x tl
```

Example 2

- Prove: if not (mem x l) then occ x l = 0

```
let rec mem x = function
| [] -> false
| hd::tl -> (hd = x) || (mem x tl)
```

```
let rec occ x = function
| [] -> 0
| hd::tl -> (if hd = x then 1 else 0) + occ x tl
```

- 1. Base case (l = [])
  - prove: if not (mem x []) then occ x [] = 0
  - if not (mem x []) then occ x [] = 0

    ≡ if not (false) then occ x [] = 0 (using definition of mem)

    ≡ if not (false) then 0 = 0 (using definition of occ)

    ≡ if true then true
  - which is always true

Example 2

- Prove: if not (mem x l) then occ x l = 0

```
let rec mem x = function
| [] -> false
| hd::tl -> (hd = x) || (mem x tl)
```

```
let rec occ x = function
| [] -> 0
| hd::tl -> (if hd = x then 1 else 0) + occ x tl
```

- 2. Inductive step (l = hd::tl)
  - inductive hypothesis: if not (mem x tl) then occ x tl
  - need to prove: if not (mem x hd::tl) then occ x hd::tl = 0
  - consider two cases:
    1. hd = x
    2. hd <> x

Example 2

- Prove: if not (mem x l) then occ x l = 0

```
let rec mem x = function
| [] -> false
| hd::tl -> (hd = x) || (mem x tl)
```

```
let rec occ x = function
| [] -> 0
| hd::tl -> (if hd = x then 1 else 0) + occ x tl
```

- 2. Inductive step (l = hd::tl)
  - inductive hypothesis: if not (mem x tl) then occ x tl
  - need to prove: if not (mem x hd::tl) then occ x hd::tl = 0
  - case (i): hd = x
    - if not (mem x hd::tl) then occ x hd::tl = 0

    ≡ if not (true || (mem x tl)) then occ x hd::tl = 0

    ≡ if false then …
  - which is always true

Example 2

- Prove: if not (mem x l) then occ x l = 0

```
let rec mem x = function
| [] -> false
| hd::tl -> (hd = x) || (mem x tl)
```

```
let rec occ x = function
| [] -> 0
| hd::tl -> (if hd = x then 1 else 0) + occ x tl
```

- 2. Inductive step (l = hd::tl)
  - inductive hypothesis: if not (mem x tl) then occ x tl
  - need to prove: if not (mem x hd::tl) then occ x hd::tl = 0
  - case (ii): hd <> x
    - if not (mem x hd::tl) then occ x hd::tl = 0

    ≡ if not (false || (mem x tl)) then occ x hd::tl = 0
    ≡ if not (mem x tl) then occ x hd::tl = 0
    ≡ if not (mem x tl) then ((if hd = x then 1 else 0) + occ x tl) = 0
    ≡ if not (mem x tl) then ((if false then 1 else 0) + occ x tl) = 0
    ≡ if not (mem x tl) then (0 + occ x tl) = 0
    ≡ if not (mem x tl) then occ x tl = 0
    (which is true by the inductive hypothesis)

Example 2

- Prove: if not (mem x l) then occ x l = 0

```
let rec mem x = function
| [] -> false
| hd::tl -> (hd = x) || (mem x tl)
```

```
let rec occ x = function
| [] -> 0
| hd::tl -> (if hd = x then 1 else 0) + occ x tl
```

- 1. Base case (l = [])
  - proved
- 2. Inductive step (l = hd::tl)
  - proved (since both cases proved)

- And therefore:
  - ∀l [if not (mem x l) then occ x l = 0]

Summary

- Mathematical induction
  - used to prove statements of the form $\forall n$ [$P(n)$]
  - where n is a natural number
  - base case, inductive step with inductive hypothesis

- Structural induction
  - used to prove properties of recursively defined objects
  - split into base case(s) & recursive case

- Exercise classes
  - Tue and Thu next week…