Structural Induction

Joseph C Osborn

April 1, 2025

Outline

Inductive "motors"

- We've seen two inductive principles so far
  - "Weak induction" over natural numbers
    - $P(0) \wedge (\forall x, P(x) \rightarrow P(x+1)) \rightarrow (\forall x, P(x))$
  - "Strong induction" over natural numbers
    - $P(0) \wedge (\forall x, (\forall y, y \leq x \rightarrow P(y)) \rightarrow P(x+1)) \rightarrow (\forall x, P(x))$
- But we can imagine others

More "motors"

- "Induction over even numbers"
  - "If P holds for an even number n, and we can show therefore P holds for n+2, then it holds for all even numbers"
- "Induction over powers of two"
  - "If P holds for a power of two x, and we can show therefore P holds for 2x, then it holds for all powers of two"
- "Induction over strings"
  - "If P holds for a string S, and we can show therefore P holds for S but with some arbitrary character appended, P holds for all strings"

Inductively Defined Structures

- Our original induction principle is nothing special
- Each of these inductive motors is defined over an inductively defined structure
  - "The next even number" is two bigger than the last
  - "The next power of two" is two times the last
  - "The next string" is one character longer
  - "The next natural number" is one bigger than the last

The Natural Numbers

- So far we've described natural numbers as an open interval from 0...
  - We could instead say "0 is a natural number, and forall natural numbers n, 1+n is a natural number".
- This framing is an *inductive definition*
  - Inductive definitions automatically provide inductive principles (motors)

Other inductive structures

- Lists
- Trees
- Graphs
- Haskell programs
- . . . and more!

Induction and Recursion

- Induction is *dual* to recursion
  - Recursion breaks down a big problem into small pieces
  - Induction builds up a big object (a value, a proof) out of small pieces
- Induction is the natural tool for proofs about computer programs
  - Whether implemented with recursion or loops

List Processing

```
length [] = 0
length (_x:l) = 1 + length l

append [] l2 = l2
append (x:l1) l2 = x:(append l1 l2)

reverse [] = []
reverse (x:l) = append (reverse l) [x]
```

Some Properties

- forall l1 l2, `length l1 + length l2 = length (append l1 l2)`
- forall l, `length l = length (reverse l)`
- forall l x, `reverse (append l [x]) = x:(reverse l)`
- forall l, `l = reverse (reverse l)`

Length-Append-Dist

- forall l1 l2, `length l1 + length l2 = length (append l1 l2)`

Length-Append-Dist

- forall l1 l2, `length l1 + length l2 = length (append l1 l2)`
- By induction on l1.

Length-Append-Dist

- forall l1 l2, `length l1 + length l2 = length (append l1 l2)`
- By induction on l1.
  - (l1 = []). WTP `length [] + length l2 = length (append [] l2)`.

Length-Append-Dist

- forall l1 l2, `length l1 + length l2 = length (append l1 l2)`
- By induction on l1.
  - (l1 = []). WTP `length [] + length l2 = length (append [] l2)`.
    - In other words, `length l2 = length l2`, which is trivially true.

Length-Append-Dist

- forall l1 l2, `length l1 + length l2 = length (append l1 l2)`
- By induction on l1.
  - (l1 = []). WTP `length [] + length l2 = length (append [] l2)`.
    - In other words, `length l2 = length l2`, which is trivially true.
  - (l1 = (x:l1')). IH: `length l1' + length l2 = length (append l1' l2)`.

Length-Append-Dist

- forall l1 l2, `length l1 + length l2 = length (append l1 l2)`
- By induction on l1.
  - (l1 = []). WTP `length [] + length l2 = length (append [] l2)`.
    - In other words, `length l2 = length l2`, which is trivially true.
  - (l1 = (x:l1')). IH: `length l1' + length l2 = length (append l1' l2)`.
    - WTP `length (x:l1') + length l2 = length (append (x:l1') l2)`.

Length-Append-Dist

- forall l1 l2, `length l1 + length l2 = length (append l1 l2)`
- By induction on l1.
  - (l1 = []). WTP `length [] + length l2 = length (append [] l2)`.
    - In other words, `length l2 = length l2`, which is trivially true.
  - (l1 = (x:l1')). IH: `length l1' + length l2 = length (append l1' l2)`.
    - WTP `length (x:l1') + length l2 = length (append (x:l1') l2)`.
    - By the definition of `append` and of length, this is:

Length-Append-Dist

- forall l1 l2, `length l1 + length l2` = `length (append l1 l2)`
- By induction on l1.
  - (l1 = []). WTP `length [] + length l2` = `length (append [] l2)`.
    - In other words, `length l2` = `length l2`, which is trivially true.
  - (l1 = (x:l1')). IH: `length l1' + length l2` = `length (append l1' l2)`.
    - WTP `length (x:l1') + length l2` = `length (append (x:l1') l2)`.
    - By the definition of append and of length, this is:
    - `1 + length l' + length l2` = `length (x:(append l1' l2)) = 1 + length (append l1' l2)`.

Length-Append-Dist

- forall l1 l2, `length l1 + length l2 = length (append l1 l2)`
- By induction on l1.
  - (l1 = []). WTP `length [] + length l2 = length (append [] l2)`.
    - In other words, `length l2 = length l2`, which is trivially true.
  - (l1 = (x:l1')). IH: `length l1' + length l2 = length (append l1' l2)`.
    - WTP `length (x:l1') + length l2 = length (append (x:l1') l2)`.
    - By the definition of append and of length, this is:
    - `1 + length l' + length l2 = length (x:(append l1' l2)) = 1 + length (append l1' l2)`.
    - We know by the IH that `length (append l1' l2)` and `length l1' + l2` are the same value, so the property is proved.

Reverse Preserves Length

- forall l, `length l = length (reverse l)`

Reverse Preserves Length

- forall l, `length l = length (reverse l)`
- By induction on l

Reverse Preserves Length

- forall l, `length l = length (reverse l)`
- By induction on l
  - (l = []). WTP `length [] = length (reverse [])`; `reverse [] = []` so this is evident.

Reverse Preserves Length

- forall l, `length l = length (reverse l)`
- By induction on l
  - (l = []). WTP `length [] = length (reverse []);` `reverse [] = []` so this is evident.
  - (l = (x:l')). IH: `length l' = length (reverse l').`

Reverse Preserves Length

- forall l, `length l = length (reverse l)`
- By induction on l
  - (l = []). WTP `length [] = length (reverse [])`; `reverse [] = []` so this is evident.
  - (l = (x:l')). IH: `length l' = length (reverse l')`.
    - WTP `length (x:l') = length (reverse (x:l'))`.

Reverse Preserves Length

- forall l, `length l = length (reverse l)`
- By induction on l
  - (l = []). WTP `length [] = length (reverse []);` `reverse [] = []` so this is evident.
  - (l = (x:l')). IH: `length l' = length (reverse l')`.
    - WTP `length (x:l') = length (reverse (x:l'))`.
    - By def'n of reverse, `1 + length l' = length (append (reverse l') [x])`.

Reverse Preserves Length

- forall l, `length l = length (reverse l)`
- By induction on l
  - (l = []). WTP `length [] = length (reverse []);` `reverse [] = []` so this is evident.
  - (l = (x:l')). IH: `length l' = length (reverse l')`.
    - WTP `length (x:l') = length (reverse (x:l'))`.
    - By def'n of reverse, `1 + length l' = length (append (reverse l') [x])`.
    - By the last property, `length (append (reverse l') [x]) = length (reverse l') + length [x] = length (reverse l') + 1`.

Reverse Preserves Length

- forall l, `length l = length (reverse l)`
- By induction on l
  - (l = []). WTP `length [] = length (reverse [])`; `reverse [] = []` so this is evident.
  - (l = (x:l')). IH: `length l' = length (reverse l')`.
    - WTP `length (x:l') = length (reverse (x:l'))`.
    - By def'n of reverse, `1 + length l' = length (append (reverse l') [x])`.
    - By the last property, `length (append (reverse l') [x]) = length (reverse l') + length [x] = length (reverse l') + 1`.
    - By the IH, `length (reverse l') = length l'`, so we have to show `1 + length l' = length l' + 1`, which is immediate by the commutativity of addition.

Reverse-Append-Single

- forall l x, `reverse (append l [x]) = x:(reverse l)`

Reverse-Append-Single

- forall l x, `reverse (append l [x]) = x:(reverse l)`
- By induction on l.

Reverse-Append-Single

- forall l x, `reverse (append l [x]) = x:(reverse l)`
- By induction on l.
  - (l = []). `reverse (append [] [x]) = reverse [x] = [x] = (x:reverse [])`.

Reverse-Append-Single

- forall l x, `reverse (append l [x]) = x:(reverse l)`
- By induction on l.
  - (l = []). `reverse (append [] [x]) = reverse [x] = [x] = (x:reverse [])`.
  - (l = (y:l')). IH: `reverse (append l' [x]) = x:(reverse l')`.

Reverse-Append-Single

- forall l x, `reverse (append l [x]) = x:(reverse l)`
- By induction on l.
  - (l = []). `reverse (append [] [x]) = reverse [x] = [x] = (x:reverse [])`.
  - (l = (y:l')). IH: `reverse (append l' [x]) = x:(reverse l')`.
    - WTP `reverse (append (y:l') [x]) = x:(reverse (y:l'))`

Reverse-Append-Single

- forall l x, `reverse (append l [x]) = x:(reverse l)`
- By induction on l.
  - (l = []). `reverse (append [] [x]) = reverse [x] = [x] = (x:reverse [])`.
  - (l = (y:l')). IH: `reverse (append l' [x]) = x:(reverse l')`.
    - WTP `reverse (append (y:l') [x]) = x:(reverse (y:l'))`
    - By def'n of append and reverse: `reverse (append (y:l') [x]) = append (reverse (append l' [x])) [y]`.

Reverse-Append-Single

- forall l x, `reverse (append l [x]) = x:(reverse l)`
- By induction on l.
  - (l = []). `reverse (append [] [x]) = reverse [x] = [x] = (x:reverse [])`.
  - (l = (y:l')). IH: `reverse (append l' [x]) = x:(reverse l')`.
    - WTP `reverse (append (y:l') [x]) = x:(reverse (y:l'))`
    - By def'n of append and reverse: `reverse (append (y:l') [x]) = append (reverse (append l' [x])) [y]`.
    - By the IH, `reverse (append l' [x]) = x:(reverse l')`, so we have `append (x:(reverse l')) [y] = x:(append (reverse l') [y])`.

Reverse-Append-Single

- forall l x, `reverse (append l [x]) = x:(reverse l)`
- By induction on l.
  - (l = []). `reverse (append [] [x]) = reverse [x] = [x] = (x:reverse [])`.
  - (l = (y:l')). IH: `reverse (append l' [x]) = x:(reverse l')`.
    - WTP `reverse (append (y:l') [x]) = x:(reverse (y:l'))`
    - By def'n of append and reverse: `reverse (append (y:l') [x]) = append (reverse (append l' [x])) [y]`.
    - By the IH, `reverse (append l' [x]) = x:(reverse l')`, so we have `append (x:(reverse l')) [y] = x:(append (reverse l') [y])`.
    - On the right side, we have `x:(reverse (y:l')) = x:(append (reverse l') [y])`, which is just our left hand side.

Reverse-Append-Single

- forall l x, `reverse (append l [x]) = x:(reverse l)`
- By induction on l.
  - (l = []). `reverse (append [] [x]) = reverse [x] = [x] = (x:reverse [])`.
  - (l = (y:l')). IH: `reverse (append l' [x]) = x:(reverse l')`.
    - WTP `reverse (append (y:l') [x]) = x:(reverse (y:l'))`
    - By def'n of append and reverse: `reverse (append (y:l') [x]) = append (reverse (append l' [x])) [y]`.
    - By the IH, `reverse (append l' [x]) = x:(reverse l')`, so we have `append (x:(reverse l')) [y] = x:(append (reverse l') [y])`.
    - On the right side, we have `x:(reverse (y:l')) = x:(append (reverse l') [y])`, which is just our left hand side.
    - So the left and right sides are equal and the theorem is proved.

Reverse-Self-Inverse

- forall l, `l = reverse (reverse l)`

Reverse-Self-Inverse

- forall l, `l = reverse (reverse l)`
- By induction on l.

Reverse-Self-Inverse

- forall l, `l = reverse (reverse l)`
- By induction on l.
  - (l = []). `reverse (reverse [])` = `reverse []` = `[]`.

Reverse-Self-Inverse

- forall l, `l = reverse (reverse l)`
- By induction on l.
  - (l = []). `reverse (reverse [])` = `reverse []` = `[]`.
  - (l = (x:l')). IH: `l'` = `reverse (reverse l')`.

Reverse-Self-Inverse

- forall l, `l = reverse (reverse l)`
- By induction on l.
  - (l = []). `reverse (reverse []) = reverse [] = []`.
  - (l = (x:l')). IH: `l' = reverse (reverse l')`.
    - WTP `(x:l') = reverse (reverse (x:l'))`.

Reverse-Self-Inverse

- forall l, `l = reverse (reverse l)`
- By induction on l.
  - (l = []). `reverse (reverse [])` = `reverse []` = `[]`.
  - (l = (x:l')). IH: `l'` = `reverse (reverse l')`.
    - WTP `(x:l')` = `reverse (reverse (x:l'))`.
    - By def'n of reverse: `reverse (reverse (x:l'))` = `reverse (append (reverse l') [x])`.

Reverse-Self-Inverse

- forall l, `l = reverse (reverse l)`
- By induction on l.
  - (l = []). `reverse (reverse [])` = `reverse []` = `[]`.
  - (l = (x:l')). IH: `l'` = `reverse (reverse l')`.
    - WTP `(x:l')` = `reverse (reverse (x:l'))`.
    - By def'n of reverse: `reverse (reverse (x:l'))` = `reverse (append (reverse l') [x])`.
    - By the previous theorem, `reverse (append (reverse l') [x])` = `x:(reverse (reverse l'))`.

Reverse-Self-Inverse

- forall l, `l = reverse (reverse l)`
- By induction on l.
  - (l = []). `reverse (reverse []) = reverse [] = []`.
  - (l = (x:l')). IH: `l' = reverse (reverse l')`.
    - WTP `(x:l') = reverse (reverse (x:l'))`.
    - By def'n of reverse: `reverse (reverse (x:l')) = reverse (append (reverse l') [x])`.
    - By the previous theorem, `reverse (append (reverse l') [x]) = x:(reverse (reverse l'))`.
    - But `reverse (reverse l')` is just `l'` by the IH, so we've shown what we are trying to prove.

Higher-Order Functions

```
map _f [] = []
map f (x:l) = (f x):(map f l)

filter _f [] = []
filter f (x:l)
  | f x = x:(filter f l)
  | otherwise = filter f l

double_all [] = []
double_all (x:l) = (x+x) : double_all l
```

- Formally state and prove these properties:
  - "The output of `map f l` has the same length as the input list"
  - "The output of `map f (append l1 l2)` is the same as `append (map f l1) (map f l2)`"
  - "`map (* 2)` is equivalent to `double_all`"
    - What does it mean for two functions to be equivalent?