Structural Induction

So far, we have seen two forms of induction: *simple induction* and *strong induction*. Both can be used to prove properties about natural numbers.

What if we want to prove properties about other things than natural numbers? For example, what if we want to prove that:

    for all lists xs :   xs ++ [] = xs

?

It is possible to prove this using induction over natural numbers, but this is not a natural thing to do. There are no natural numbers present anywhere in the statement of the property above!

Structural induction is a proof technique that can be used to prove properties about other kinds of things than natural numbers. These things have to have a certain kind of recursive structure; this is why it is called *structural* induction.

The new kind of things we can prove properties about are for example:

- properties about lists

- properties about recursive expressions

- properties about trees

- ...

In fact, structural induction can be used to prove properties about any *recursive datatype*.

Anatomy of a proof by Structural Induction

A proof by structural induction has basically the same shape as a proof by simple induction: There is a base case (but there can be several), and there is a step case (but in general there can be several), in which we assume we have already proved the property for smaller things.

Here is an example. Given the definition of ++:

```
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

we would like to prove that xs ++ [] = xs, for any list xs.

To prove: xs ++ [] = xs, for any list xs

Proof: by structural induction.

Let P(xs) = “xs ++ [] = xs”.

base case: P([]) = “[] ++ [] = []”. Follows from the defn. of ++.

step case: P(as) => P(a:as)
assume: P(as) = “as ++ [] = as” (I.H.)
show: P(a:as) = “(a:as) ++ [] = a:as”

```
(a:as) ++ [] = a : (as ++ [])        (defn. of ++)
             = a : as                (I.H.)
```

which is what we had to prove. □

From the above, we can see that structural induction over lists involve two cases: a base case and a step case. In the base case, we may assume that the list is empty. In the step case, we may assume that the list is non-empty, and moreover (the I.H.) that the property we want to prove actually holds for the tail of that non-empty list.

Structural induction over lists involves one base case and one step case, because the list datatype has one non-recursive constructor (namely []), and one recursive constructor (namely :). In general, structural induction has one base case for each non-recursive constructor, and one step case for each recursive constructor, in which we may assume that the property already has been proven for the recursive arguments of that constructor.

Here is an example. Consider the following datatype.

```
data Expr
    = Num Integer
     | Add Expr Expr
     | Mul Expr Expr
```

And the following function:

```
swap (Num n)    = Num n
swap (Add a b)  = Add (swap b) (swap a)
swap (Mul a b)   = Mul (swap b) (swap a)
```

Suppose we would like to prove the following:

To prove: swap (swap x) = x, for all expressions x.

Proof: by structural induction.

Let P(x) = “swap (swap x) = x”.

base case: P(Num n) = “swap (swap (Num n)) = Num n”.

```
  swap (swap (Num n))
= swap (Num n)                  (defn. swap)
= Num n                         (defn. swap)
```

step case 1: (P(a) /\ P(b)) => P(Add a b)
assume: P(a) = “swap (swap a) = a”
P(b) = “swap (swap b) = b”   (I.H.)
show: P(Add a b) = “swap (swap (Add a b)) = Add a b”

```
  swap (swap (Add a b))
= swap (Add (swap b) (swap a))              (defn. swap)
= Add (swap (swap a)) (swap (swap b))       (defn. swap)
= Add a b                                   (I.H. * 2)
```

step case 2: (P(a) /\ P(b)) => P(Mul a b)
assume: P(a) = “swap (swap a) = a”
P(b) = “swap (swap b) = b”   (I.H.)
show: P(Add a b) = “swap (swap (Mul a b)) = Mul a b”

```
  swap (swap (Mul a b))
= swap (Mul (swap b) (swap a))              (defn. swap)
```

```
    = Mul (swap (swap a)) (swap (swap b))      (defn. swap)
    = Mul a b                                   (I.H. * 2)
```

which is what we had to prove. □

Expr has one non-recursive constructor (Num), which means that there is one base case (Num n). Expr has two recursive constructors (Add and Mul), which means that there are two step cases (Add a b,Mul a b). In the step cases, we may assume that property already has been proven for the recursive arguments (a and b in this case).

What follows are many example proofs using structural induction. First, some proofs over lists, and then some proofs over general recursive datatypes.

More examples of proofs of properties over lists

++ adds together the lengths of the lists

Consider the following definition:

```
length []     = 0
length (x:xs) = 1 + length xs
```

To prove: length (xs ++ ys) = length xs + length ys, for all lists xs,ys.

Proof: by structural induction.

Let P(xs) = “forall ys . length (xs ++ ys) = length xs + length ys”.

base case: P([]) = “forall ys . length ([] ++ ys) = length [] + length ys”.

```
  length ([] ++ ys)
= length ys                 (defn. ++)
= 0 + length ys
= length [] + length ys     (defn. length)
```

step case: P(as) => P(a:as).
assume: P(as)   = “forall zs . length (as ++ zs) = length as + length zs”   (I.H.)
show: P(a:as) = “forall ys . length ((a:as) ++ ys) = length (a:as) + length ys”

```
  length ((a:as) ++ ys)
= length (a:(as ++ ys))     (defn. ++)
= 1 + length (as ++ ys)              (defn. length)
= 1 + length as + length ys          (I.H.)
= length (a:as) + length ys          (defn. length)
```

which is what we had to prove. □

Note: If you try to prove this by induction over ys, you will get stuck. Try it!

last is an element of the list

Consider the following definitions:

```
last [x]      = x
last (x:xs) = last xs

x `elem` []        = False
x `elem` (y:ys) = x == y || (x `elem` ys)
```

To prove: last xs `elem` xs, for all non-empty lists xs.

Proof: by structural induction.

Let P(xs) = “last xs `elem` xs”.

base case: P([a]) = “last [a] `elem` [a]”.

```
  last [a] `elem` [a]
= a `elem` [a]                    (defn. last)
= a == a || a `elem` []           (defn. elem)
= True || ..                      (reflexivity ==)
= True
```

step case: P(as) => P(a:as), when as is non-empty.
assume: P(as)   = “last as `elem` as” (I.H.)
as is non-empty
show: P(a:as) = “last (a:as) `elem` (a:as)”

```
  last (a:as) `elem` (a:as)
= last as `elem` (a:as)           (defn. last, as is non-empty)
= last as == a || last as `elem` as       (defn. elem)
= .. || True                              (I.H.)
= True
```

which is what we had to prove. □

Note: Since we only want to prove something about non-empty lists, the base case becomes a list of length 1, and in the step case we may assume that as is non-empty.

++ is associative

To prove: xs ++ (ys ++ zs) = (xs ++ ys) ++ zs, for all lists xs,ys,zs.

Proof: by structural induction.

Let P(xs) = “forall ys, zs . xs ++ (ys ++ zs) = (xs ++ ys) ++ zs”.

base case: P([]) = “forall ys, zs . [] ++ (ys ++ zs) = ([] ++ ys) ++ zs”.

[] ++ (ys ++ zs)
= ys ++ zs (defn. ++)
= ([] ++ ys) ++ zs (defn. ++)

step case: P(as) => P(a:as).
assume: P(as) = “forall vs, ws . as ++ (vs ++ ws) = (as ++ vs) ++ ws” (I.H.)
show: P(a:as) = “forall ys, zs . (a:as) ++ (ys ++ zs) = ((a:as) ++ ys) ++ zs”

(a:as) ++ (ys ++ zs)
= a : (as ++ (ys ++ zs)) (defn. ++)
= a : ((as ++ ys) ++ zs) (I.H.)
= ((a:(as ++ ys)) ++ zs) (defn. ++)
= ((a:as) ++ ys) ++ zs) (defn. ++)

which is what we had to prove. □

Note: If you try to prove this by induction over ys or zs, you will get stuck. Try it!

reverse swaps the arguments to ++

Consider the following definition of reverse:

```
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

To prove: reverse (xs ++ ys) = reverse ys ++ reverse xs, for all lists xs, ys.

Proof: by structural induction.

Let P(xs) = “forall ys . reverse (xs ++ ys) = reverse ys ++ reverse xs”.

base case: P([]) = “forall ys . reverse ([] ++ ys) = reverse ys ++ reverse []”.

```
  reverse ([] ++ ys)
= reverse ys                  (defn. reverse)
= reverse ys ++ []            (++/[] lemma)
= reverse ys ++ reverse []
```

step case: P(as) => P(a:as).

assume: P(as)   = “forall zs . reverse (as ++ zs) = reverse zs ++ reverse as” (I.H.)

show: P(a:as) = “forall ys . reverse ((a:as) ++ ys) = reverse ys ++ reverse (a:as)”

```
  reverse ((a:as) ++ ys)
= reverse (a : (as ++ ys))                (defn. ++)
= reverse (as ++ ys) ++ [a]               (defn. reverse)
= (reverse ys ++ reverse as) ++ [a]       (I.H.)
= reverse ys ++ (reverse as ++ [a])       (++ is associative)
= reverse ys ++ reverse (a:as)            (defn. reverse)
```

which is what we had to prove. □

Note: We need to use the fact that ++ is associative in the proof. If we had not known this on beforehand, we would have had to state that and also prove it, before we would have been able to continue.

---

reverse is its own inverse function

Using the above property (which we call the reverse/++ lemma here), we can prove a property about reverse alone.

To prove: reverse (reverse xs) = xs, for all lists xs.

Proof: by structural induction.

Let P(xs) = “reverse (reverse xs) = xs”.

base case: P([]) = “reverse (reverse []) = []”.

```
  reverse (reverse [])
= reverse []                 (defn. reverse)
= []                         (defn. reverse)
```

step case: P(as) => P(a:as).
assume: P(as)   = “reverse (reverse as) = as” (I.H.)
show: P(a:as) = “reverse (reverse (a:as)) = a:as”

```
  reverse (reverse (a:as))                (since xs=a:as)
= reverse (reverse as ++ [a])             (defn. reverse)
= reverse [a] ++ reverse (reverse as)(reverse/++ lemma)
= [a] ++ reverse (reverse as)             (defn. reverse)
= [a] ++ as                               (I.H.)
= a:as                                    (defn. ++)
```

which is what we had to prove. □

Note: We use the reverse/++ lemma in the proof. If we had not known this to be true already, we would have had to state it and then prove it. Normally, we may have to find out such a lemma is needed, state it, and prove it, in order to prove the thing we were actually interested in.

qreverse is correct

There is a much better way to implement reverse. Consider the following definition:

```
qreverse []     ys = ys
qreverse (x:xs) ys = qreverse xs (x:ys)
```

We would like to show that qreverse xs [] = reverse xs. However, this cannot be shown by structural induction directly. We need to generalize the statement to something that is more easily proved by induction:

To prove: qreverse xs ys = reverse xs ++ ys, for all lists xs, ys.

Proof: by structural induction.

Let P(xs) = “forall ys . qreverse xs ys = reverse xs ++ ys”

base case: P([]) = “forall ys . qreverse [] ys = reverse [] ++ ys”.

```
  qreverse [] ys              (since xs=[])
= ys                          (defn. qreverse)
= reverse [] ++ ys            (defn. reverse, ++)
```

step case: P(as) => P(a:as).
assume: P(as)   = “forall zs . qreverse as zs = reverse as ++ zs” (I.H.)
show: P(a:as) = “forall ys. qreverse (a:as) ys = reverse (a:as) ++ ys”

```
  qreverse (a:as) ys
= qreverse as (a:ys)                  (defn. qreverse)
= reverse as ++ (a:ys)        (I.H., zs=a:ys)
= reverse as ++ ([a] ++ ys)           (defn. ++)
= (reverse as ++ [a]) ++ ys           (++ is associative)
= reverse (a:as) ++ ys        (defn. reverse)
```

which is what we had to prove. □

Note: The I.H. still forall-quantifies the second list, which we renamed to zs. This is important, because when we use the I.H. in the proof, that list zs is equal to a:ys. If we had just used ys in the I.H., we could not have used the I.H.

take/drop lemma

Consider the following definitions:

```
take n _ | n<=0= []
take n []       = []
take n (x:xs)   = x : take (n-1) xs

drop n xs | n<=0 = xs
drop n []        = []
drop n (x:xs)    = drop (n-1) xs
```

To prove: take n xs ++ drop n xs = xs, for any integer n and any list n

Proof: by structural induction.

Let P(xs) = “forall n . take n xs ++ drop n xs = xs”.

base case: P(xs) = “forall n . take n [] ++ drop n [] = []”.

```
  take n [] ++ drop n []
= [] ++ []                  (defn. take, drop)
= []                        (defn. ++)
```

step case: P(as) => P(a:as).
assume: P(as) = “forall k . take k as ++ drop k as = as” (I.H.)
show: P(a:as) = “forall n . take n (a:as) ++ drop n (a:as) = a:as”

We do a case split on n:

case 1: n <= 0.

```
  take n (a:as) ++ drop n (a:as)
= [] ++ (a:as)                      (defn. take, drop)
= a:as                              (defn. ++)
```

case 2: n > 0.

```
  take n (a:as) ++ drop n (a:as)
= (a : take (n-1) as) ++ drop (n-1) as        (defn. take, drop; n>0)
= a : (take (n-1) as ++ drop (n-1) as)        (defn. ++)
= a : as                                      (I.H., k=n-1)
```

which was what we had to prove. □

Note 1: The I.H. still forall-quantifies the integer in the property, which we renamed to k. This is important, because when we use the I.H. in the proof, that integer k is equal to n-1. If we had just used n in the I.H., we could not have used the I.H.

Note 2: It is also possible to prove the above by *integer induction* over n. Integer induction proves something about all integers. The base case is n<=0. The step case is the normal step case we know from simple induction.

The resulting proof of the take/drop lemma looks similar to the above, except that we do a case split on xs in the step case: the case where xs=[] and the case where xs=a:as.

Example structural induction proofs over general recursive datatypes

a relationship between height and size of trees

Consider the following datatype and functions:

```
data Tree = Empty | Node Tree Tree

size (Empty)     = 0
size (Node p q) = 1 + size p + size q

height Empty       = 0
height (Node p q) = 1 + (height p `max` height q)
```

To prove: size $x \le 2^{\text{height } x} - 1$, for all trees x.

Proof: by structural induction.

Let P(x) = “size $x \le 2^{\text{height } x} - 1$”.

base case: P(Empty) = “size Empty $\le 2^{\text{height Empty}} - 1$”.

size Empty
$= 0$ (defn. size)
$\le 1 - 1$
$= 2^{0} - 1$
$= 2^{\text{height Empty}} - 1$ (defn. height)

step case: (P(p) /\ P(q)) => P(Node p q).
assume: P(p) = “size $p \le 2^{\text{height } p} - 1$”
P(q) = “size $q \le 2^{\text{height } q} - 1$” (I.H.)
show: P(Node p q) = “size (Node p q) $\le 2^{\text{height (Node p q)}} - 1$”

size (Node p q)
$= 1 + \text{size } p + \text{size } q$ (defn. size)
$\le 1 + (2^{\text{height } p} - 1) + (2^{\text{height } q} - 1)$ (I.H. * 2)
$= 2^{\text{height } p} + 2^{\text{height } q} - 1$

$\le 2 \cdot (2^{\text{height p `max` height q}}) - 1$ ($v+w \le 2\cdot(v$ `max` $w)$)
$= 2^{1 + (\text{height p `max` height q})} - 1$

$= 2^{\text{height (Node p q)}} - 1$ (defn. height)

which is what we had to prove. □

swapping does not affect the value

Consider the following functions over expressions:

```
eval (Num n)   = n
eval (Add a b) = eval a + eval b
eval (Mul a b)  = eval a * eval b

swap (Num n)   = Num n
swap (Add a b) = Add (swap b) (swap a)
swap (Mul a b)  = Mul (swap b) (swap a)
```

To prove: eval (swap x) = eval x, for all expressions x

Proof: by structural induction.

Let P(x) = “eval (swap x) = eval x”.

base case: P(Num n) = “eval (swap (Num n)) = eval (Num n)”.

```
  eval (swap (Num n))
= eval (Num n)                    (defn. swap)
```

step case 1: ((P(a) /\ P(b)) => P(Add a b).
assume: P(a) = “eval (swap a) = eval a”
P(b) = “eval (swap b) = eval b”   (I.H.)
show: P(Add a b) = “eval (swap (Add a b)) = eval (Add a b)”

```
  eval (swap (Add a b))
= eval (Add (swap b) (swap a))        (defn. swap)
= eval (swap b) + eval (swap a)       (defn. eval)
= eval b + eval a                     (I.H. * 2)
= eval a + eval b                     (+ commutative)
= eval (Add a b)                      (defn. eval)
```

step case 2: ((P(a) /\ P(b)) => P(Mul a b).
assume: P(a) = “eval (swap a) = eval a”
P(b) = “eval (swap b) = eval b”   (I.H.)
show: P(Add a b) = “eval (swap (Mul a b)) = eval (Mul a b)”

```
  eval (swap (Mul a b))
= eval (Mul (swap b) (swap a))        (defn. swap)
= eval (swap b) + eval (swap a)        (defn. eval)
= eval b * eval a                      (I.H. * 2)
= eval a * eval b                      (* commutative)
= eval (Mul a b)                       (defn. eval)
```

which is what we had to prove. □

simplification does not affect the value

Consider the following simplification function for expressions:

```
simp (Num n)   = Num n
simp (Add a b) = Add (simp a) (simp b)
simp (Mul a b)  | a == Num 1 = simp b
                | b == Num 1 = simp a
                | otherwise    = Mul (simp a) (simp b)
```

To prove: eval (simp x) = eval x, for all expressions x

Proof: by structural induction.

Let P(x) = “eval (simp x) = eval x”.

base case: P(Num n).

```
  eval (simp (Num n))
= n                              (defn. simp, eval)
= eval (Num n)                   (defn. eval)
```

step case 1: (P(a) /\ P(b)) => P(Add a b).

assume: P(a) = “eval (simp a) = eval a”
P(b) = “eval (simp b) = eval b”   (I.H.)
show: P(Add a b) = “eval (simp (Add a b)) = eval (Add a b)”

```
  eval (simp (Add a b))
= eval (Add (simp a) (simp b))          (defn. simp)
= eval (simp a) + eval (simp b)         (defn. eval)
= eval a + eval b                       (I.H. * 2)
= eval (Add a b)                        (defn. eval)
= eval x
```

step case 2: (P(a) /\ P(b)) => P(Mul a b).

assume: P(a) = “eval (simp a) = eval a”
P(b) = “eval (simp b) = eval b” (I.H.)
show: P(Mul a b) = “eval (simp (Mul a b)) = eval (Mul a b)”

We perform a case analysis on the Mul-cases of simp:

case 1: a = Num 1.

```
  eval (simp (Mul a b))
= eval (simp b)                     (defn. simp)
= eval b                            (I.H.)
= eval (Num 1) * eval b             (1 * z = z)
= eval (Mul (Num 1) b)              (defn. eval)
= eval (Mul a b)
```

case 2: a /= Num 1, b = Num 1.

```
  eval (simp (Mul a b))
= eval (simp a)                     (defn. simp)
= eval a                            (I.H.)
= eval a * eval (Num 1)             (z * 1 = z)
= eval (Mul a (Num 1))              (defn. eval)
= eval (Mul a b)
```

case 3: a /= Num 1, b /= Num 1.

```
  eval (simp (Mul a b))
= eval (Mul (simp a) (simp b))(defn. simp)
= eval (simp a) * eval (simp b)     (defn. eval)
= eval a * eval b                   (I.H. * 2)
= eval (Mul a b)                    (defn. eval)
```

which is what we had to prove. □

isZero approximates eval = 0

Consider the following function that tries to quickly see if an expression equals 0 or not:

```
isZero (Num n)   = n==0
isZero (Add a b) = isZero a ∧ isZero b
isZero (Mul a b)  = isZero a ∨ isZero b
```

To prove: isZero x => eval x = 0, for all expressions x.

Proof: by structural induction.

Let P(x) = “isZero x => eval x = 0”.

base case: P(Num n) = “isZero (Num n) => eval (Num n) = 0”.

1. We may assume isZero (Num n). This means that:

```
  isZero (Num n)
⇒ n == 0                    (defn. isZero)
```

```
2.      eval (Num n)            (since x=Num n)
      = n                       (defn. eval)
      = 0                       (1.)
```

step case 1: (P(a) /\ P(b)) => P(Add a b).
assume: P(a) = “isZero a => eval a = 0”
P(b) = “isZero b => eval b = 0” (I.H.)
show: P(Add a b) = “isZero (Add a b) => eval (Add a b) = 0”

1. We may assume isZero (Add a b). This means that:

```
  isZero (Add a b)
⇒ isZero a ∧ isZero b                (defn. isZero)
```

2. By (1.) and (I.H.) we have:

```
eval a = 0
eval b = 0
```

3. So:

```
  eval (Add a b)
```

= eval a + eval b (defn. eval)
= 0 + 0 (2.)
= 0

step case 2: (P(a) /\ P(b)) => P(Mul a b).
assume: P(a) = “isZero a => eval a = 0”
P(b) = “isZero b => eval b = 0” (I.H.)
show: P(Mul a b) = “isZero (Mul a b) => eval (Mul a b) = 0”

1. We may assume isZero (Mul a b). This means that:

isZero (Mul a b)
⇒ isZero a \/ isZero b (defn. isZero)

2. By (1.) and (I.H.) we have:

eval a = 0 \/ eval b = 0

3. eval (Mul a b)
= eval a * eval b (defn. eval)

Now, we do a case split on (2.):

case 1: eval a = 0.

eval a * eval b
= 0 * eval b
= 0

case 2: eval b = 0.

eval a * eval b
= eval a * 0
= 0

which is what we had to prove. □

Note: When we prove something with an implication in it, we have to be careful what we know in each case and what we have to show in order to use the I.H.

When we prove something of the shape A => B, we may assume A and proceed to prove B.

When we want to use something of the shape A => B, we first have to show that A holds, and then we also know that B holds.