---

**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 on xs.

**base case:** xs = [].

```
xs ++ [] = [] ++ []        (since xs=[])
         = []              (defn. of ++)
         = xs
```

**step case:** xs = a:as.

1. We may assume that  as ++ [] = as  (I.H.)

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

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 on x.

**base case:** x=Num n.

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

**step case 1:** x=Add a b.

1. We may assume

```
      swap (swap a) = a
and   swap (swap b) = b.              (I.H.)
```

2.
```
  swap (swap x)
= swap (swap (Add a b))                      (since x=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)
= x
```

**step case 2:** x=Mul a b.

1. We may assume

```
        swap (swap a) = a
and     swap (swap b) = b.              (I.H.)
```

2.
```
    swap (swap x)
= swap (swap (Mul a b))                 (since x=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)
= x
```

which is what we had to prove. □

Expr has one non-recursive constructor (Num), which means that there is one base case (x=Num n). Expr has two recursive constructors (Add and Mul), which means that there are two step cases (x=Add a b, x=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 over xs.

**base case:** xs = [].

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

**step case:** xs = a:as.

1. We may assume that length (as ++ ys) = length as + length ys, for all lists ys. (I.H.)

2.
```
  length (xs ++ ys)
= length ((a:as) ++ ys)          (since xs=a:as)
= 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)
= length xs + length ys
```

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 over xs.

**base case:** xs = [a].

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

**step case:** xs = a:as, as is non-empty.

1. We may assume that  last as `elem` as. (I.H.)

2.
```
   last xs `elem` xs
= last (a:as) `elem` (a:as)          (since xs=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 over xs.

**base case:** xs = [].

```
   xs ++ (ys ++ zs)
= [] ++ (ys ++ zs)          (since xs=[])
= ys ++ zs                  (defn. ++)
= ([] ++ ys) ++ zs          (defn. ++)
= (xs ++ ys) ++ zs
```

**step case:** xs = a:as.

1. We may assume that  as ++ (ys ++ zs) = (as ++ ys) ++ zs,  for all lists ys and zs. (I.H.)

2.
```
   xs ++ (ys ++ zs)
= (a:as) ++ (ys ++ zs)          (since xs=a:as)
= a : (as ++ (ys ++ zs))        (defn. ++)
= a : ((as ++ ys) ++ zs)        (I.H.)
= ((a:(as ++ ys)) ++ zs)        (defn. ++)
= ((a:as) ++ ys) ++ zs)         (defn. ++)
= (xs ++ ys) ++ zs)
```

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 over xs.

**base case:** xs = [].

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

**step case:** xs = a:as.

1. We may assume that  reverse (as ++ ys) = reverse ys ++ reverse as. (I.H.)

2.
```
   reverse (xs ++ ys)
= reverse ((a:as) ++ ys)                (since xs=a:as)
= 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)
= reverse ys ++ reverse xs
```

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 over xs.

**base case:** xs = [].

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

**step case:** xs = a:as.

1. We may assume that  reverse (reverse as) = as. (I.H.)

2.
```
   reverse (reverse xs)
= 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. ++)
= xs
```

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 over xs.

**base case:** xs = [].

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

**step case:** xs = a:as.

1. We may assume that  qreverse as zs = reverse as ++ zs,  for all lists zs. (I.H.)

2.
```
   qreverse xs ys
= qreverse (a:as) ys              (since xs=a:as)
= 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)
= reverse xs ++ ys
```

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 on xs.

**base case:** xs = [].

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

**step case:** xs = a:as.

1. We may assume that take k as ++ drop k as, for any integer k (I.H.)

2. We do a case split on n:

**case 1:** n <= 0.

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

**case 2:** n > 0.

```
  take n xs ++ drop n xs
= take n (a:as) ++ drop n (a:as)            (since xs=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)
= xs

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 ≤ $2^{\text{height x}}$ - 1, for all trees x.

**Proof:** by structural induction over x.

**base case:** x = Empty.

| | |
|---|---|
| size x | |
| = size Empty | (since x=Empty) |
| = 0 | (defn. size) |
| ≤ 1 - 1 | |
| = $2^0$ - 1 | |
| = $2^{\text{height Empty}}$ - 1 | (defn. height) |
| = $2^{\text{height x}}$ - 1 | |

**step case:** x = Node p q.

1. We may assume that (I.H.):

size p ≤ $2^{\text{height p}}$ - 1
and size q ≤ $2^{\text{height q}}$ - 1

2.

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

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

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

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

$= 2^{\text{height x}} - 1$

which is what we had to prove. □

---

**swapping does not affect the value**

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

**Proof:** by structural induction on x.

**base case:** x = Num n.

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

**step case 1:** x = Add a b.

1. We may assume that (I.H.):

```
        eval (swap a) = eval a
and     eval (swap b) = eval b
```

2.
```
   eval (swap x)
= eval (swap (Add a b))                (since x=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)
= eval x
```

**step case 2:** x = Mul a b.

1. We may assume that (I.H.):

```
        eval (swap a) = eval a
and     eval (swap b) = eval b
```

2.
```
   eval (swap x)
= eval (swap (Mul a b))                (since x=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)
= eval x

which is what we had to prove. □

---

**simplification 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

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 on x.

**base case:** x = Num n.

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

**step case 1:** x = Add a b.

1. We may assume that (I.H.):

```
        eval (simp a) = eval a
and   eval (simp b) = eval b
```

2.
```
   eval (simp x)
= eval (simp (Add a b))                (since x=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:** x = Mul a b.

1. We may assume that (I.H.):

```
        eval (simp a) = eval a
and     eval (simp b) = eval b
```

2.
```
  eval (simp x)
= eval (simp (Mul a b))                (since x=Mul a b)
```

We now 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 x
```

**case 2:** 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 x
```

**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)
= eval x
```

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 over x.

**base case:** x = Num n.

1. We may assume isZero x. This means that:

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

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

**step case 1:** x = Add a b.

1. We may assume (I.H.):

```
      isZero a => eval a = 0
and   isZero b => eval b = 0
```

2. We may assume isZero x. This means that:

```
   isZero x
⇒ isZero (Add a b)          (since x=Add a b)
⇒ isZero a ∧ isZero b                (defn. isZero)
```

4. By (1.) and (2.) we have:

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

3. eval x
= eval (Add a b) (since x=Add a b)
= eval a + eval b (defn. eval)
= 0 + 0 (4.)
= 0

**step case 2:** x = Mul a b.

1. We may assume (I.H.):

isZero a => eval a = 0
and isZero b => eval b = 0

2. We may assume isZero x. This means that:

isZero x
⇒ isZero (Mul a b) (since x=Mul a b)
⇒ isZero a ∨ isZero b (defn. isZero)

4. By (1.) and (2.) we have:

eval a = 0 ∨ eval b = 0

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

Now, we do a case split on (4.):

**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.