Script generated by TTT

Title: Nipkow: Info2 (05.11.2013)

Date: Tue Nov 05 15:30:19 CET 2013

Duration: 95:34 min

Pages: 109

Primitive recursion on lists:

```
f []       = base   -- base case
f (x : xs) = rec    -- recursive case
```

- *base*: no call of *f*
- *rec*: only call(s) *f xs*

f may have additional parameters.

Finding primitive recursive definitions

Example

```
concat :: [[a]] -> [a]
```

Beyond primitive recursion: Multiple arguments

Example

```
zip :: [a] -> [b] -> [(a,b)]
```

General recursion: Quicksort

Example

```
quicksort :: Ord a => [a] -> [a]
```

General recursion: Quicksort

Example

```
quicksort :: Ord a => [a] -> [a]
quicksort []  =  []
```

General recursion: Quicksort

Example

```
quicksort :: Ord a => [a] -> [a]
quicksort []  =  []
quicksort (x:xs)  =
  quicksort below ++ [x] ++ quicksort above
```


General recursion: Quicksort

Example

```
quicksort :: Ord a => [a] -> [a]
quicksort []  =  []
quicksort (x:xs)  =
  quicksort below ++ [x] ++ quicksort above
    where
      below = [y | y <- xs, y <= x]
      above = [y | y <- xs, x < y]
```

Accumulating parameter

Idea: Result is accumulated in parameter and returned later

Accumulating parameter

Idea: Result is accumulated in parameter and returned later

Example: list of all (maximal) ascending sublists in a list

Accumulating parameter

Idea: Result is accumulated in parameter and returned later

Example: list of all (maximal) ascending sublists in a list

```
ups [3,0,2,3,2,4] =
```

Accumulating parameter

Idea: Result is accumulated in parameter and returned later

Example: list of all (maximal) ascending sublists in a list

```
ups [3,0,2,3,2,4] = [[3], [0,2,3], [2,4]]
```

Accumulating parameter

Idea: Result is accumulated in parameter and returned later

Example: list of all (maximal) ascending sublists in a list

```
ups [3,0,2,3,2,4] = [[3], [0,2,3], [2,4]]

ups2 :: Ord a => [a] -> [a] -> [[a]]
```

Accumulating parameter

Idea: Result is accumulated in parameter and returned later

Example: list of all (maximal) ascending sublists in a list

```
ups [3,0,2,3,2,4] = [[3], [0,2,3], [2,4]]

ups2 :: Ord a => [a] -> [a] -> [[a]]
-- 1st param: input list
-- 2nd param: partial ascending sublist
```

Accumulating parameter

Idea: Result is accumulated in parameter and returned later

Example: list of all (maximal) ascending sublists in a list

```
ups [3,0,2,3,2,4] = [[3], [0,2,3], [2,4]]

ups2 :: Ord a => [a] -> [a] -> [[a]]
-- 1st param: input list
-- 2nd param: partial ascending sublist (reversed)
```

Accumulating parameter

Idea: Result is accumulated in parameter and returned later

Example: list of all (maximal) ascending sublists in a list

```
ups [3,0,2,3,2,4] = [[3], [0,2,3], [2,4]]

ups2 :: Ord a => [a] -> [a] -> [[a]]
-- 1st param: input list
-- 2nd param: partial ascending sublist (reversed)
```

Accumulating parameter

Idea: Result is accumulated in parameter and returned later

Example: list of all (maximal) ascending sublists in a list

```
ups [3,0,2,3,2,4] = [[3], [0,2,3], [2,4]]

ups2 :: Ord a => [a] -> [a] -> [[a]]
-- 1st param: input list
-- 2nd param: partial ascending sublist (reversed)
ups2 (x:xs) (y:ys)
  | x >= y     =  ups2 xs (x:y:ys)
  | otherwise  =  reverse (y:ys) : ups2 (x:xs) []
ups2 (x:xs) []  = ups2 xs [x]
ups2 []     ys  = [reverse ys]
```

Accumulating parameter

Idea: Result is accumulated in parameter and returned later

Example: list of all (maximal) ascending sublists in a list

```
ups [3,0,2,3,2,4] = [[3], [0,2,3], [2,4]]

ups2 :: Ord a => [a] -> [a] -> [[a]]
-- 1st param: input list
-- 2nd param: partial ascending sublist (reversed)
ups2 (x:xs) (y:ys)
  | x >= y     =  ups2 xs (x:y:ys)
  | otherwise  =  reverse (y:ys) : ups2 (x:xs) []
ups2 (x:xs) []  = ups2 xs [x]
ups2 []     ys  = [reverse ys]

ups :: Ord a => [a] -> [[a]]
ups xs = ups2 xs []
```


Accumulating parameter

Idea: Result is accumulated in parameter and returned later

Example: list of all (maximal) ascending sublists in a list

```
ups [3,0,2,3,2,4] = [[3], [0,2,3], [2,4]]

ups2 :: Ord a => [a] -> [a] -> [[a]]
-- 1st param: input list
-- 2nd param: partial ascending sublist (reversed)
ups2 (x:xs) (y:ys)
  | x >= y     =  ups2 xs (x:y:ys)
  | otherwise  =  reverse (y:ys) : ups2 (x:xs) []
ups2 (x:xs) []  = ups2 xs [x]
ups2 []     ys  = [reverse ys]

ups :: Ord a => [a] -> [[a]]
ups xs = ups2 xs []
```

Accumulating parameter

Idea: Result is accumulated in parameter and returned later

Example: list of all (maximal) ascending sublists in a list

```
ups [3,0,2,3,2,4] = [[3], [0,2,3], [2,4]]

ups2 :: Ord a => [a] -> [a] -> [[a]]
-- 1st param: input list
-- 2nd param: partial ascending sublist (reversed)
ups2 (x:xs) (y:ys)
  | x >= y     =  ups2 xs (x:y:ys)
  | otherwise  =  reverse (y:ys) : ups2 (x:xs) []
ups2 (x:xs) []  = ups2 xs [x]
ups2 []     ys  = [reverse ys]

ups :: Ord a => [a] -> [[a]]
ups xs = ups2 xs []
```

Convention

Identifiers of list type end in ‘s’:

xs, ys, zs, ...

Mutual recursion

Example

```
even :: Int -> Bool
even n  =  n == 0 || n > 0 && odd (n-1) || odd (n+1)

odd :: Int -> Bool
odd n  =  n /= 0 && (n > 0 && even (n-1) || even (n+1))
```

Scoping by example

```
x = y + 5
y = x + 1 where x = 7
f y = y + x

> f 3
```

Scoping by example

```
x = y + 5
y = x + 1 where x = 7
f y = y + x

> f 3
16
```

Scoping by example

```
x = y + 5
y = x + 1 where x = 7
f y = y + x

> f 3
16
```

Binding occurrence

Scoping by example

```
x = y + 5
y = x + 1 where x = 7
f y = y + x

> f 3
16
```

Binding occurrence
Bound occurrence
Scope of binding

Scoping by example

```
x = y + 5
y = x + 1 where x = 7
f y = y + x

> f 3
16
```

Binding occurrence
Bound occurrence
Scope of binding

Scoping by example

```
x = y + 5
y = x + 1 where x = 7
f y = y + x

> f 3
16
```

Binding occurrence
Bound occurrence
Scope of binding

Scoping by example

```
x = y + 5
y = x + 1 where x = 7
f y = y + x
```

```
> f 3
16
```

Binding occurrence
Bound occurrence
Scope of binding

Scoping by example

```
x = y + 5
y = x + 1 where x = 7
f y = y + x
```

```
> f 3
16
```

Binding occurrence
Bound occurrence
Scope of binding

Scoping by example

Summary:

- Order of definitions is irrelevant
- Parameters and `where`-defs are local to each equation

5. Proofs

Aim

Guarentee functional (I/O) properties of software

Aim

Guarentee functional (I/O) properties of software

- Testing can guarantee properties for some inputs.
- Mathematical proof can guarantee properties for all inputs.

Aim

Guarentee functional (I/O) properties of software

- Testing can guarantee properties for some inputs.
- Mathematical proof can guarantee properties for all inputs.

QuickCheck is good, proof is better

Beware of bugs in the above code;
I have only proved it correct, not tried it.

Donald E. Knuth, 1977

5.1 Proving properties

What do we prove?

Equations `e1 = e2`

5.1 Proving properties

What do we prove?

Equations `e1 = e2`

How do we prove them?

By using defining equations `f p = t`

A first, simple example

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

A first, simple example

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

Proof of `[1,2] ++ [] = [1] ++ [2]`:

A first, simple example

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

Proof of `[1,2] ++ [] = [1] ++ [2]`:

```
  1:2:[] ++ []
```

A first, simple example

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

Proof of `[1,2] ++ [] = [1] ++ [2]`:

```
1:2:[] ++ []
= 1 : (2:[] ++ [])
```

A first, simple example

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

Proof of `[1,2] ++ [] = [1] ++ [2]`:

```
1:2:[] ++ []
= 1 : (2:[] ++ [])     -- by def of ++
```

A first, simple example

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

Proof of `[1,2] ++ [] = [1] ++ [2]`:

```
1:2:[] ++ []
= 1 : (2:[] ++ [])     -- by def of ++
= 1 : 2 : ([] ++ [])   -- by def of ++
= 1 : 2 : []           -- by def of ++
```

A first, simple example

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

Proof of `[1,2] ++ [] = [1] ++ [2]`:

```
1:2:[] ++ []
= 1 : (2:[] ++ [])     -- by def of ++
= 1 : 2 : ([] ++ [])   -- by def of ++
= 1 : 2 : []           -- by def of ++
= 1 : ([] ++ 2:[])     -- by def of ++
```

A first, simple example

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

Proof of `[1,2] ++ [] = [1] ++ [2]`:

```
1:2:[] ++ []
= 1 : (2:[] ++ [])     -- by def of ++
= 1 : 2 : ([] ++ [])   -- by def of ++
= 1 : 2 : []           -- by def of ++
= 1 : ([] ++ 2:[])     -- by def of ++
= 1:[] ++ 2:[]         -- by def of ++
```

A first, simple example

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

Proof of `[1,2] ++ [] = [1] ++ [2]`:

```
1:2:[] ++ []
= 1 : (2:[] ++ [])     -- by def of ++
= 1 : 2 : ([] ++ [])   -- by def of ++
= 1 : 2 : []           -- by def of ++
= 1 : ([] ++ 2:[])     -- by def of ++
= 1:[] ++ 2:[]         -- by def of ++
```


Observation: first used equations from left to right (ok),

A first, simple example

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

Proof of `[1,2] ++ [] = [1] ++ [2]`:

```
1:2:[] ++ []
= 1 : (2:[] ++ [])     -- by def of ++
= 1 : 2 : ([] ++ [])   -- by def of ++
= 1 : 2 : []           -- by def of ++
= 1 : ([] ++ 2:[])     -- by def of ++
= 1:[] ++ 2:[]         -- by def of ++
```

Observation: first used equations from left to right (ok),
then from right to left (strange!)

A more natural proof of `[1,2] ++ [] = [1] ++ [2]`:

```
1:2:[] ++ []
= 1 : (2:[] ++ [])     -- by def of ++
= 1 : 2 : ([] ++ [])   -- by def of ++
= 1 : 2 : []           -- by def of ++

1:[] ++ 2:[]
= 1 : ([] ++ 2:[])     -- by def of ++
= 1 : 2 : []           -- by def of ++
```

A more natural proof of `[1,2] ++ [] = [1] ++ [2]`:

```
1:2:[] ++ []
= 1 : (2:[] ++ [])      -- by def of ++
= 1 : 2 : ([] ++ [])    -- by def of ++
= 1 : 2 : []            -- by def of ++

1:[] ++ 2:[]
= 1 : ([] ++ 2:[])      -- by def of ++
= 1 : 2 : []            -- by def of ++
```

Proofs of `e1 = e2` are often better presented
as two reductions to some expression e:

```
e1 = ...  = e
e2 = ...  = e
```

Fact If an equation does not contain any variables, it can be proved by evaluating both sides separately and checking that the result is identical.

Fact If an equation does not contain any variables, it can be proved by evaluating both sides separately and checking that the result is identical.

But how to prove equations with variables, for example *associativity* of ++:

```
(xs ++ ys) ++ zs = xs ++ (ys ++ zs)
```

Properties of recursive functions are proved by induction

Properties of recursive functions are proved by induction

Induction on natural numbers: see Diskrete Strukturen

Properties of recursive functions are proved by induction

Induction on natural numbers: see Diskrete Strukturen

Induction on lists: here and now

Structural induction on lists

To prove property `P(xs)` for all finite lists `xs`

Structural induction on lists

To prove property `P(xs)` for all finite lists `xs`

Base case: Prove `P([])` and

Structural induction on lists

To prove property `P(xs)` for all finite lists `xs`

Base case: Prove `P([])` and

Induction step: Prove `P(xs)` implies `P(x:xs)`

Structural induction on lists

To prove property `P(xs)` for all finite lists `xs`

Base case: Prove `P([])` and

Induction step: Prove `P(xs)` implies `P(x:xs)`

↑ *induction hypothesis*

Structural induction on lists

To prove property `P(xs)` for all finite lists `xs`

Base case: Prove `P([])` and

Induction step: Prove `P(xs)` implies `P(x:xs)`

↑ *induction hypothesis* (*IH*)

↑ new variable `x`

Structural induction on lists

To prove property `P(xs)` for all finite lists `xs`

Base case: Prove `P([])` and

Induction step: Prove `P(xs)` implies `P(x:xs)`

↑ *induction hypothesis* (*IH*)

↑ new variable `x`

This is called *structural induction* on `xs`.

Structural induction on lists

To prove property `P(xs)` for all finite lists `xs`

Base case: Prove `P([])` and

Induction step: Prove `P(xs)` implies `P(x:xs)`

↑ *induction hypothesis* (*IH*)   ↑ new variable `x`

This is called *structural induction* on `xs`.
It is a special case of induction on the length of `xs`.

Example: associativity of ++

Lemma app_assoc: `(xs ++ ys) ++ zs = xs ++ (ys ++ zs)`

Example: associativity of ++

Lemma app_assoc: `(xs ++ ys) ++ zs = xs ++ (ys ++ zs)`
Proof by structural induction on `xs`

Example: associativity of ++

Lemma app_assoc: `(xs ++ ys) ++ zs = xs ++ (ys ++ zs)`
Proof by structural induction on `xs`

Base case:
To show: `([] ++ ys) ++ zs = [] ++ (ys ++ zs)`

Example: associativity of ++

Lemma app_assoc: `(xs ++ ys) ++ zs = xs ++ (ys ++ zs)`
Proof by structural induction on `xs`

Base case:
To show: `([] ++ ys) ++ zs = [] ++ (ys ++ zs)`

```
([] ++ ys) ++ zs
```


Example: associativity of ++

Lemma app_assoc: `(xs ++ ys) ++ zs = xs ++ (ys ++ zs)`
Proof by structural induction on `xs`

Base case:
To show: `([] ++ ys) ++ zs = [] ++ (ys ++ zs)`


```
([] ++ ys) ++ zs
= ys ++ zs            -- by def of ++
```


Example: associativity of ++

Lemma app_assoc: `(xs ++ ys) ++ zs = xs ++ (ys ++ zs)`
Proof by structural induction on `xs`

Base case:
To show: `([] ++ ys) ++ zs = [] ++ (ys ++ zs)`


```
([] ++ ys) ++ zs
= ys ++ zs            -- by def of ++
= [] ++ (ys ++ zs)    -- by def of ++
```


Example: associativity of ++

Lemma app_assoc: `(xs ++ ys) ++ zs = xs ++ (ys ++ zs)`
Proof by structural induction on `xs`

Base case:
To show: `([] ++ ys) ++ zs = [] ++ (ys ++ zs)`


```
([] ++ ys) ++ zs
= ys ++ zs            -- by def of ++
= [] ++ (ys ++ zs)    -- by def of ++
```

Induction step:
To show: `((x:xs) ++ ys) ++ zs = (x:xs) ++ (ys ++ zs)`

Example: associativity of ++

Lemma app_assoc: `(xs ++ ys) ++ zs = xs ++ (ys ++ zs)`
Proof by structural induction on `xs`

Base case:
To show: `([] ++ ys) ++ zs = [] ++ (ys ++ zs)`

```
 ([] ++ ys) ++ zs
 = ys ++ zs            -- by def of ++
 = [] ++ (ys ++ zs)    -- by def of ++
```

Induction step:
To show: `((x:xs) ++ ys) ++ zs = (x:xs) ++ (ys ++ zs)`

```
 ((x:xs) ++ ys) ++ zs
```

Example: associativity of ++

Lemma app_assoc: `(xs ++ ys) ++ zs = xs ++ (ys ++ zs)`
Proof by structural induction on `xs`

Base case:
To show: `([] ++ ys) ++ zs = [] ++ (ys ++ zs)`

```
 ([] ++ ys) ++ zs
 = ys ++ zs            -- by def of ++
 = [] ++ (ys ++ zs)    -- by def of ++
```

Induction step:
To show: `((x:xs) ++ ys) ++ zs = (x:xs) ++ (ys ++ zs)`

```
 ((x:xs) ++ ys) ++ zs
 = (x : (xs ++ ys)) ++ zs   -- by def of ++
```

Example: associativity of ++

Lemma app_assoc: `(xs ++ ys) ++ zs = xs ++ (ys ++ zs)`
Proof by structural induction on `xs`

Base case:
To show: `([] ++ ys) ++ zs = [] ++ (ys ++ zs)`

```
 ([] ++ ys) ++ zs
 = ys ++ zs            -- by def of ++
 = [] ++ (ys ++ zs)    -- by def of ++
```

Induction step:
To show: `((x:xs) ++ ys) ++ zs = (x:xs) ++ (ys ++ zs)`

```
 ((x:xs) ++ ys) ++ zs
 = (x : (xs ++ ys)) ++ zs   -- by def of ++
 = x : ((xs ++ ys) ++ zs)   -- by def of ++
```

Example: associativity of ++

Lemma app_assoc: `(xs ++ ys) ++ zs = xs ++ (ys ++ zs)`
Proof by structural induction on `xs`

Base case:
To show: `([] ++ ys) ++ zs = [] ++ (ys ++ zs)`

```
 ([] ++ ys) ++ zs
 = ys ++ zs            -- by def of ++
 = [] ++ (ys ++ zs)    -- by def of ++
```

Induction step:
To show: `((x:xs) ++ ys) ++ zs = (x:xs) ++ (ys ++ zs)`

```
 ((x:xs) ++ ys) ++ zs
 = (x : (xs ++ ys)) ++ zs   -- by def of ++
 = x : ((xs ++ ys) ++ zs)   -- by def of ++
 = x : (xs ++ (ys ++ zs))   -- by IH
```

Example: associativity of ++

Lemma app_assoc: `(xs ++ ys) ++ zs = xs ++ (ys ++ zs)`
Proof by structural induction on `xs`

Base case:
To show: `([] ++ ys) ++ zs = [] ++ (ys ++ zs)`

```
([] ++ ys) ++ zs
= ys ++ zs             -- by def of ++
= [] ++ (ys ++ zs)     -- by def of ++
```

Induction step:
To show: `((x:xs) ++ ys) ++ zs = (x:xs) ++ (ys ++ zs)`

```
((x:xs) ++ ys) ++ zs
= (x : (xs ++ ys)) ++ zs   -- by def of ++
= x : ((xs ++ ys) ++ zs)   -- by def of ++
= x : (xs ++ (ys ++ zs))   -- by IH
(x:xs) ++ (ys ++ zs)
```


Example: associativity of ++

Lemma app_assoc: `(xs ++ ys) ++ zs = xs ++ (ys ++ zs)`
Proof by structural induction on `xs`

Base case:
To show: `([] ++ ys) ++ zs = [] ++ (ys ++ zs)`

```
([] ++ ys) ++ zs
= ys ++ zs             -- by def of ++
= [] ++ (ys ++ zs)     -- by def of ++
```

Induction step:
To show: `((x:xs) ++ ys) ++ zs = (x:xs) ++ (ys ++ zs)`

```
((x:xs) ++ ys) ++ zs
= (x : (xs ++ ys)) ++ zs   -- by def of ++
= x : ((xs ++ ys) ++ zs)   -- by def of ++
= x : (xs ++ (ys ++ zs))   -- by IH
(x:xs) ++ (ys ++ zs)
```


```
= x : (xs ++ (ys ++ zs))   -- by def of ++
```

Induction template

Lemma `P(xs)`

Induction template

Lemma `P(xs)`

Proof by structural induction on `xs`

Base case:
To show: `P([])`

Induction template

Lemma `P(xs)`
Proof by structural induction on `xs`

Base case:
To show: `P([])`

Proof of `P([])`

Induction template

Lemma `P(xs)`
Proof by structural induction on `xs`

Base case:
To show: `P([])`

Proof of `P([])`

Induction step:
To show: `P(x:xs)`

Proof of `P(x:xs)` using IH `P(xs)`

Example: `length` of `++`

Lemma `length(xs ++ ys) = length xs + length ys`

Example: `length` of `++`

Lemma `length(xs ++ ys) = length xs + length ys`
Proof by structural induction on `xs`

Example: length of ++

Lemma `length(xs ++ ys) = length xs + length ys`
Proof by structural induction on xs
Base case:

Example: length of ++

Lemma `length(xs ++ ys) = length xs + length ys`
Proof by structural induction on xs
Base case:
To show: `length ([] ++ ys) = length [] + length ys`

```
 length ([] ++ ys)
 = length ys              -- by def of ++
 length [] + length ys
 = 0 + length ys          -- by def of length
 = length ys
```

Example: length of ++

Lemma `length(xs ++ ys) = length xs + length ys`
Proof by structural induction on xs
Base case:
To show: `length ([] ++ ys) = length [] + length ys`

```
 length ([] ++ ys)
 = length ys              -- by def of ++
 length [] + length ys
 = 0 + length ys          -- by def of length
 = length ys
```

Induction step:
To show: `length((x:xs)++ys) = length(x:xs) + length ys`

Example: length of ++

Lemma `length(xs ++ ys) = length xs + length ys`
Proof by structural induction on xs
Base case:
To show: `length ([] ++ ys) = length [] + length ys`

```
 length ([] ++ ys)
 = length ys              -- by def of ++
 length [] + length ys
 = 0 + length ys          -- by def of length
 = length ys
```

Induction step:
To show: `length((x:xs)++ys) = length(x:xs) + length ys`

```
 length((x:xs) ++ ys)
 = length(x : (xs ++ ys))      -- by def of ++
```

Example: length of ++

Lemma `length(xs ++ ys) = length xs + length ys`
Proof by structural induction on xs
Base case:
To show: `length ([] ++ ys) = length [] + length ys`

```
length ([] ++ ys)
= length ys               -- by def of ++
length [] + length ys
= 0 + length ys           -- by def of length
= length ys
```

Induction step:
To show: `length((x:xs)++ys) = length(x:xs) + length ys`

```
length((x:xs) ++ ys)
= length(x : (xs ++ ys))      -- by def of ++
= 1 + length(xs ++ ys)        -- by def of length
```

Example: length of ++

Lemma `length(xs ++ ys) = length xs + length ys`
Proof by structural induction on xs
Base case:
To show: `length ([] ++ ys) = length [] + length ys`

```
length ([] ++ ys)
= length ys               -- by def of ++
length [] + length ys
= 0 + length ys           -- by def of length
= length ys
```

Induction step:
To show: `length((x:xs)++ys) = length(x:xs) + length ys`

```
length((x:xs) ++ ys)
= length(x : (xs ++ ys))      -- by def of ++
= 1 + length(xs ++ ys)        -- by def of length
= 1 + length xs + length ys   -- by IH
length(x:xs) + length ys
= 1 + length xs + length ys   -- by def of length
```

Example: length of ++

Lemma `length(xs ++ ys) = length xs + length ys`
Proof by structural induction on xs
Base case:
To show: `length ([] ++ ys) = length [] + length ys`

```
length ([] ++ ys)
= length ys               -- by def of ++
length [] + length ys
= 0 + length ys           -- by def of length
= length ys
```

Induction step:
To show: `length((x:xs)++ys) = length(x:xs) + length ys`

```
length((x:xs) ++ ys)
= length(x : (xs ++ ys))      -- by def of ++
= 1 + length(xs ++ ys)        -- by def of length
= 1 + length xs + length ys   -- by IH
```

Example: length of ++

Lemma `length(xs ++ ys) = length xs + length ys`
Proof by structural induction on xs
Base case:
To show: `length ([] ++ ys) = length [] + length ys`

```
length ([] ++ ys)
= length ys               -- by def of ++
length [] + length ys
= 0 + length ys           -- by def of length
= length ys
```

Induction step:
To show: `length((x:xs)++ys) = length(x:xs) + length ys`

```
length((x:xs) ++ ys)
= length(x : (xs ++ ys))      -- by def of ++
= 1 + length(xs ++ ys)        -- by def of length
= 1 + length xs + length ys   -- by IH
length(x:xs) + length ys
= 1 + length xs + length ys   -- by def of length
```

Example: reverse of ++

Lemma `reverse(xs ++ ys) = reverse ys ++ reverse xs`

Example: reverse of ++

Lemma `reverse(xs ++ ys) = reverse ys ++ reverse xs`
Proof by structural induction on `xs`

Example: reverse of ++

Lemma `reverse(xs ++ ys) = reverse ys ++ reverse xs`
Proof by structural induction on `xs`
Base case:

Example: reverse of ++

Lemma `reverse(xs ++ ys) = reverse ys ++ reverse xs`
Proof by structural induction on `xs`
Base case:
To show: `reverse ([] ++ ys) = reverse ys ++ reverse []`

```
reverse ([] ++ ys)
= reverse ys                -- by def of ++
reverse ys ++ reverse []
= reverse ys ++ []          -- by def of reverse
= reverse ys                -- by
```

Example: `reverse` of `++`

Lemma `reverse(xs ++ ys) = reverse ys ++ reverse xs`
Proof by structural induction on `xs`
Base case:
To show: `reverse ([] ++ ys) = reverse ys ++ reverse []`

```
reverse ([] ++ ys)
```

Example: `reverse` of `++`

Lemma `reverse(xs ++ ys) = reverse ys ++ reverse xs`
Proof by structural induction on `xs`
Base case:
To show: `reverse ([] ++ ys) = reverse ys ++ reverse []`

```
reverse ([] ++ ys)
= reverse ys               -- by def of ++
reverse ys ++ reverse []
= reverse ys ++ []         -- by def of reverse
= reverse ys               -- by Lemma app_Nil2
```

Example: `reverse` of `++`

Lemma `reverse(xs ++ ys) = reverse ys ++ reverse xs`
Proof by structural induction on `xs`
Base case:
To show: `reverse ([] ++ ys) = reverse ys ++ reverse []`

```
reverse ([] ++ ys)
= reverse ys               -- by def of ++
reverse ys ++ reverse []
= reverse ys ++ []         -- by def of reverse
= reverse ys               -- by Lemma app_Nil2
```

Lemma app_Nil2: `xs ++ [] = xs`

Example: `reverse` of `++`

Lemma `reverse(xs ++ ys) = reverse ys ++ reverse xs`
Proof by structural induction on `xs`
Base case:
To show: `reverse ([] ++ ys) = reverse ys ++ reverse []`

```
reverse ([] ++ ys)
= reverse ys               -- by def of ++
reverse ys ++ reverse []
= reverse ys ++ []         -- by def of reverse
= reverse ys               -- by Lemma app_Nil2
```

Lemma app_Nil2: `xs ++ [] = xs`
Proof exercise

Induction step:

To show: `reverse((x:xs)++ys) = reverse ys ++ reverse(x:xs)`

Induction step:

To show: `reverse((x:xs)++ys) = reverse ys ++ reverse(x:xs)`

```
reverse((x:xs) ++ ys)
= reverse(x : (xs ++ ys))            -- by def of ++
```

Induction step:

To show: `reverse((x:xs)++ys) = reverse ys ++ reverse(x:xs)`

```
reverse((x:xs) ++ ys)
= reverse(x : (xs ++ ys))            -- by def of ++
= reverse(xs ++ ys) ++ [x]           -- by def of reverse
```

Induction step:

To show: `reverse((x:xs)++ys) = reverse ys ++ reverse(x:xs)`

```
reverse((x:xs) ++ ys)
= reverse(x : (xs ++ ys))            -- by def of ++
= reverse(xs ++ ys) ++ [x]           -- by def of reverse
= (reverse ys ++ reverse xs) ++ [x]  -- by IH
```

Induction step:

To show: `reverse((x:xs)++ys) = reverse ys ++ reverse(x:xs)`

```
reverse((x:xs) ++ ys)
= reverse(x : (xs ++ ys))            -- by def of ++
= reverse(xs ++ ys) ++ [x]           -- by def of reverse
= (reverse ys ++ reverse xs) ++ [x]  -- by IH

reverse ys ++ reverse(x:xs)
= reverse ys ++ (reverse xs ++ [x])  -- by def of reverse
```

Induction step:

To show: `reverse((x:xs)++ys) = reverse ys ++ reverse(x:xs)`

```
reverse((x:xs) ++ ys)
= reverse(x : (xs ++ ys))            -- by def of ++
= reverse(xs ++ ys) ++ [x]           -- by def of reverse
= (reverse ys ++ reverse xs) ++ [x]  -- by IH
= reverse ys ++ (reverse xs ++ [x])  -- by Lemma app_assoc
reverse ys ++ reverse(x:xs)
= reverse ys ++ (reverse xs ++ [x])  -- by def of reverse
```

Proof heuristic

- Try QuickCheck

Proof heuristic

- Try QuickCheck
- Try to evaluate both sides to common term

Proof heuristic

- Try QuickCheck
- Try to evaluate both sides to common term
- Try induction

Proof heuristic

- Try QuickCheck
- Try to evaluate both sides to common term
- Try induction
  - Base case: reduce both sides to a common term using function defs and lemmas
  - Induction step: reduce both sides to a common term using function defs, IH and lemmas

Proof heuristic

- Try QuickCheck
- Try to evaluate both sides to common term
- Try induction
  - Base case: reduce both sides to a common term using function defs and lemmas
  - Induction step: reduce both sides to a common term using function defs, IH and lemmas
- If base case or induction step fails:
  conjecture, prove and use new lemmas

Proof heuristic

- Try QuickCheck
- Try to evaluate both sides to common term

Example: `reverse` of `++`

Lemma `reverse(xs ++ ys) = reverse ys ++ reverse xs`
Proof by structural induction on `xs`
Base case:
To show: `reverse ([] ++ ys) = reverse ys ++ reverse []`

```
reverse ([] ++ ys)
= reverse ys              -- by def of ++
reverse ys ++ reverse []
= reverse ys ++ []        -- by def of reverse
= reverse ys              -- by Lemma app_Nil2
```

Lemma app_Nil2: `xs ++ [] = xs`

Proof heuristic

- Try QuickCheck
- Try to evaluate both sides to common term
- Try induction
  - Base case: reduce both sides to a common term using function defs and lemmas
  - Induction step: reduce both sides to a common term using function defs, IH and lemmas
- If base case or induction step fails: conjecture, prove and use new lemmas