Script **generated by TTT**

Title: Nipkow: Info2 (31.10.2014)

Date: Fri Oct 31 07:29:47 GMT 2014

Duration: 86:07 min

Pages: 136

General recursion: Quicksort

Example

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


General recursion: Quicksort

Example

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


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]
```

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]
```

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]
```


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

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

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]]
```


```
ups :: Ord 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]]

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]]

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


```
ups2 :: Ord a => [a] -> [a] -> [[a]]
-- 1st param: input 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] = [[3], [0,2,3], [2,4]]

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

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]]

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

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]]

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

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

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]]

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

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

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

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

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

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

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

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

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

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

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

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

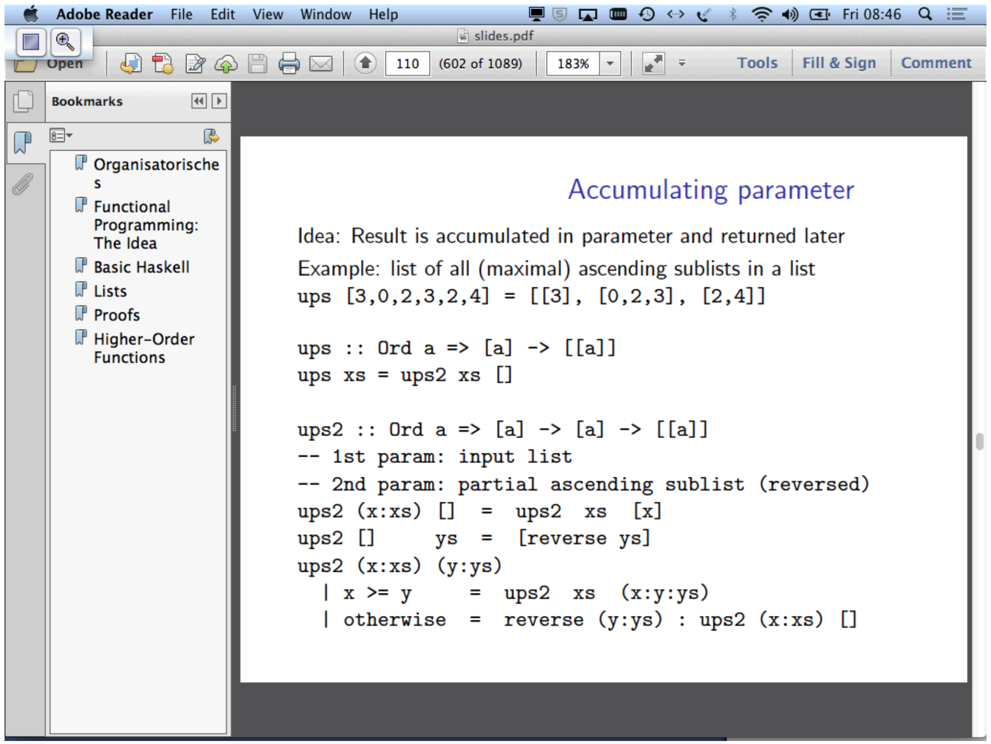
Accumulating parameter

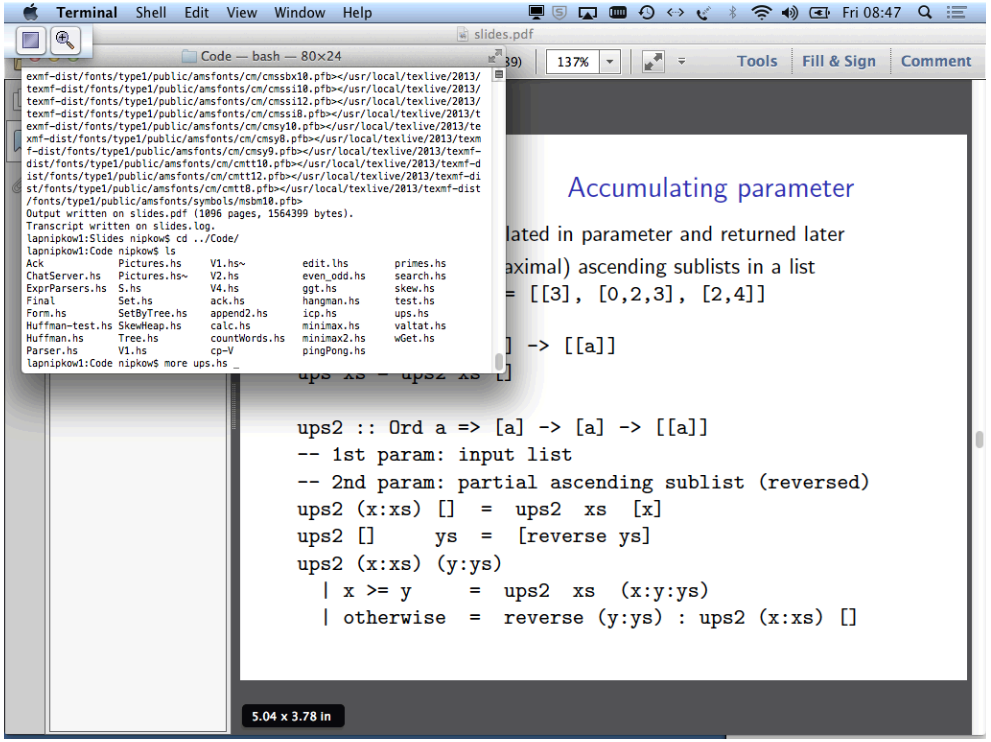
Idea: Result is accumulated in parameter and returned later

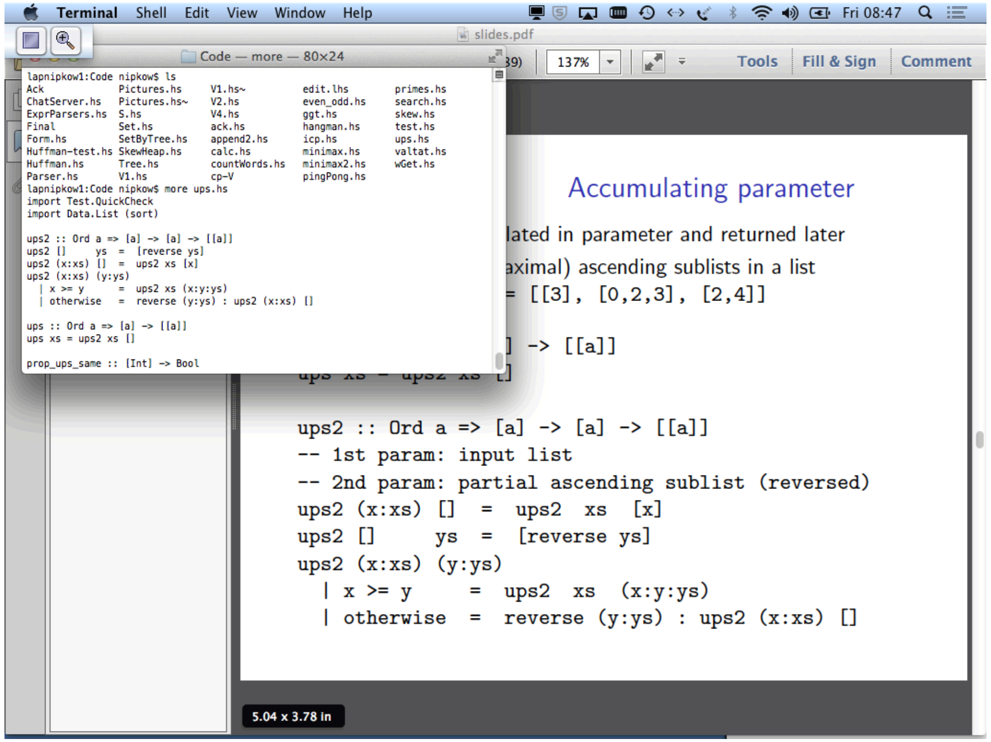
Example: list of all (maximal) ascending sublists in a list

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

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

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

How can we quickCheck the result of ups?







How can we `quickCheck` the result of `ups`?

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

```
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

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

TUM gegen KIT!

TUM gegen KIT!

Die Wettbewerbsaufgaben
der kommenden *n* Übungsblätter
werden auch am KIT gestellt

TUM gegen KIT!

Die Wettbewerbsaufgaben
der kommenden *n* Übungsblätter
werden auch am KIT gestellt
(*Programmierparadigmen*, 5. Sem., Prof. Snelting)

TUM gegen KIT!

Die Wettbewerbsaufgaben
der kommenden *n* Übungsblätter
werden auch am KIT gestellt
(*Programmierparadigmen*, 5. Sem., Prof. Snelting)
und werden gemeinsam bewertet.

Wo studieren die besseren Programmierer?

TUM gegen KIT!

Die Wettbewerbsaufgaben
der kommenden *n* Übungsblätter
werden auch am KIT gestellt
(*Programmierparadigmen*, 5. Sem., Prof. Snelting)
und werden gemeinsam bewertet.

Wo studieren die besseren Programmierer?
TUM oder KIT?

TUM gegen KIT!

Die Wettbewerbsaufgaben
der kommenden *n* Übungsblätter
werden auch am KIT gestellt
(*Programmierparadigmen*, 5. Sem., Prof. Snelting)
und werden gemeinsam bewertet.

Wo studieren die besseren Programmierer?
TUM oder KIT?
Zeigen Sie, dass TUM TOP ist!

5. Proofs

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

5.1 Proving properties

5.1 Proving properties

What do we prove?

Equations `e1 = e2`

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 ++
```


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 ++
```

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 ++
```


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 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 ++
```


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.

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)`

↑

induction hypothesis (*IH*)

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*)

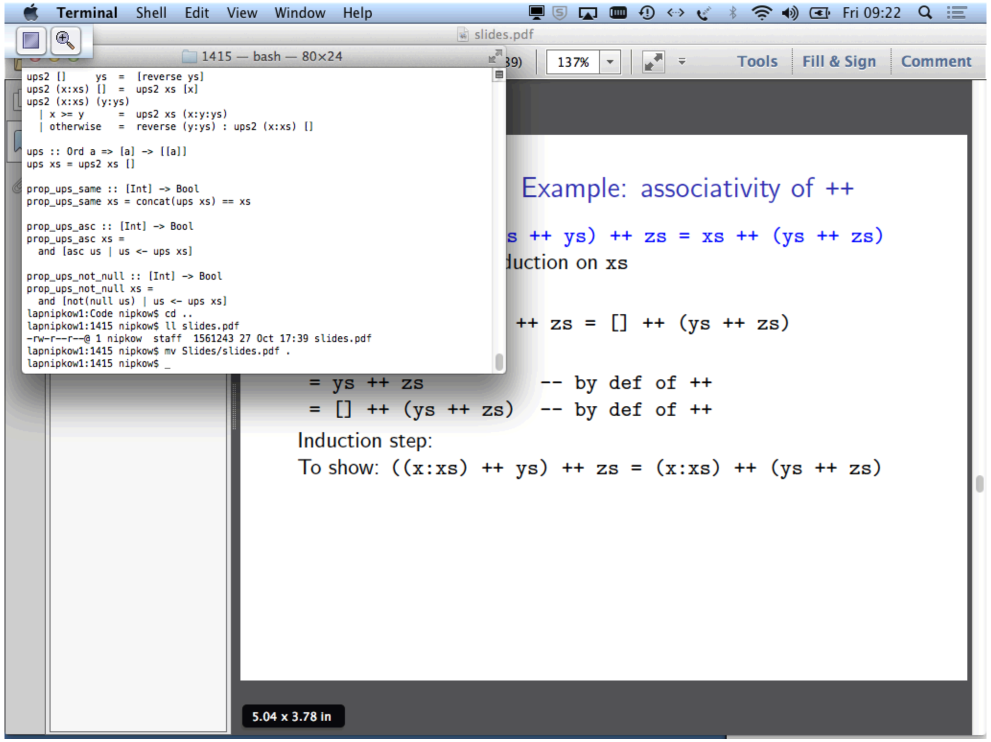
Structural induction on lists

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

One and the same fixed `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

One and the same fixed `xs`!

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

One and the same fixed `xs`!

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)`
Proof by structural induction on `xs`

Base case:

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:

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

Induction step:
IH: `([] ++ ys) ++ zs = [] ++ (ys ++ zs)`
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:
IH: `([] ++ ys) ++ zs = [] ++ (ys ++ zs)`
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:
IH: `([] ++ ys) ++ zs = [] ++ (ys ++ zs)`
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:
IH: `([] ++ ys) ++ zs = [] ++ (ys ++ zs)`
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:
IH: `([] ++ ys) ++ zs = [] ++ (ys ++ zs)`
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)
```

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 step:

Induction template

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

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

Proof of `P([])`

Induction step:

IH: `P(xs)`
To show: `P(x:xs)`

Induction template

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

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

Proof of `P([])`

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

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

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

```
 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
```

Induction step:
IH: `length(xs ++ ys) = length xs + length ys`

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

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

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

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

Induction step:
IH: `length(xs ++ ys) = length xs + length ys`
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
```

Induction step:
IH: `length(xs ++ ys) = length xs + length ys`
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:
To show: `reverse ([] ++ ys) = reverse ys ++ reverse []`

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

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)
= 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`
Proof exercise

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

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


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


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


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

Induction step:

IH: `reverse(xs ++ ys) = reverse ys ++ reverse xs`

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:

IH: `reverse(xs ++ ys) = reverse ys ++ reverse xs`

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:

IH: `reverse(xs ++ ys) = reverse ys ++ reverse xs`

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
```

Induction step:

IH: `reverse(xs ++ ys) = reverse ys ++ reverse xs`

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:

IH: `reverse(xs ++ ys) = reverse ys ++ reverse xs`

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
- Try induction
  - Base case: reduce both sides to a common term using function defs 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

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
- 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

Example: `reverse` of `++`

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

Two further tricks

- Proof by cases
- Generalization

Example: proof by cases

`rem x []  =  []`
`rem x (y:ys) | x==y        =  rem x ys`
`             | otherwise   =  y : rem x ys`

Example: proof by cases

`rem x []  =  []`
`rem x (y:ys) | x==y        =  rem x ys`
`             | otherwise   =  y : rem x ys`

Lemma `rem z (xs ++ ys) = rem z xs ++ rem z ys`

`rem x []  =  []`
`rem x (y:ys) | x==y        =  rem x ys`
`             | otherwise   =  y : rem x ys`

Induction step:
IH: `rem z (xs ++ ys) = rem z xs ++ rem z ys`
To show: `rem z ((x:xs)++ys) = rem z (x:xs) ++ rem z ys`

`rem x []  =  []`
`rem x (y:ys) | x==y        =  rem x ys`
`             | otherwise   =  y : rem x ys`

Induction step:
IH: `rem z (xs ++ ys) = rem z xs ++ rem z ys`
To show: `rem z ((x:xs)++ys) = rem z (x:xs) ++ rem z ys`

Proof by cases:

Case `z == x`:

```
 rem z ((x:xs) ++ ys)
 = rem z (xs ++ ys)          -- by def of ++ and rem
 = rem z xs ++ rem z ys      -- by IH
```

```
rem x []     = []
rem x (y:ys) | x==y      = rem x ys
             | otherwise = y : rem x ys
```

Induction step:

IH: `rem z (xs ++ ys) = rem z xs ++ rem z ys`

To show: `rem z ((x:xs)++ys) = rem z (x:xs) ++ rem z ys`

Proof by cases:

Case `z == x`:

```
rem z ((x:xs) ++ ys)
= rem z (xs ++ ys)         -- by def of ++ and rem
= rem z xs ++ rem z ys     -- by IH
```

```
rem x []     = []
rem x (y:ys) | x==y      = rem x ys
             | otherwise = y : rem x ys
```

Induction step:

IH: `rem z (xs ++ ys) = rem z xs ++ rem z ys`

To show: `rem z ((x:xs)++ys) = rem z (x:xs) ++ rem z ys`

Proof by cases:

Case `z == x`:

```
rem z ((x:xs) ++ ys)
= rem z (xs ++ ys)         -- by def of ++ and rem
= rem z xs ++ rem z ys     -- by IH
rem z (x:xs) ++ rem z ys
```

```
rem x []     = []
rem x (y:ys) | x==y      = rem x ys
             | otherwise = y : rem x ys
```

Induction step:

IH: `rem z (xs ++ ys) = rem z xs ++ rem z ys`

To show: `rem z ((x:xs)++ys) = rem z (x:xs) ++ rem z ys`

Proof by cases:

Case `z == x`:

```
rem z ((x:xs) ++ ys)
= rem z (xs ++ ys)         -- by def of ++ and rem
= rem z xs ++ rem z ys     -- by IH
rem z (x:xs) ++ rem z ys
= rem z xs ++ rem z ys     -- by def of rem
```

```
rem x []     = []
rem x (y:ys) | x==y      = rem x ys
             | otherwise = y : rem x ys
```

Induction step:

IH: `rem z (xs ++ ys) = rem z xs ++ rem z ys`

To show: `rem z ((x:xs)++ys) = rem z (x:xs) ++ rem z ys`

Proof by cases:

Case `z == x`:

```
rem z ((x:xs) ++ ys)
= rem z (xs ++ ys)         -- by def of ++ and rem
= rem z xs ++ rem z ys     -- by IH
rem z (x:xs) ++ rem z ys
= rem z xs ++ rem z ys     -- by def of rem
```

Case `z /= x`:

```
rem z ((x:xs) ++ ys)
= x : rem z (xs ++ ys)         -- by def of ++ and rem
= x : (rem z xs ++ rem z ys)   -- by IH
rem z (x:xs) ++ rem z ys
```

Proof by cases

Works just as well for `if-then-else`,

Proof by cases

Works just as well for `if-then-else`,

Proof by cases

Works just as well for `if-then-else`, for example

```
rem x []      =  []
rem x (y:ys)  =  if x == y then rem x ys
                 else y : rem x ys
```

Inefficiency of `reverse`

```
reverse [1,2,3]
```

Inefficiency of `reverse`

```
reverse [1,2,3]
```