Example

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

Example

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

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

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

Accumulating parameter

Idea: Result is accumulated in parameter and returned later

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

```haskell
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
\[
\text{ups } [3,0,2,3,2,4] = [[3], [0,2,3], [2,4]]
\]

\[
\text{ups } : \text{Ord } a \Rightarrow [a] \rightarrow [[a]]
\]

\[
\text{ups } x = \text{ups2 } x \ []
\]

\[
\text{ups2 } : \text{Ord } a \Rightarrow [a] \rightarrow [a] \rightarrow [[a]]
\]

-- 1st param: input list
Idea: Result is accumulated in parameter and returned later
Example: list of all (maximal) ascending sublists in a list
\[
\text{ups} [3,0,2,3,2,4] = [[3], [0,2,3], [2,4]]
\]
\[
\text{ups} :: \text{Ord} \ a \Rightarrow [a] \Rightarrow [[a]]
\]
\[
\text{ups} \ x\text{s} = \text{ups2} \ x\text{s} \ []
\]
\[
\text{ups2} :: \text{Ord} \ a \Rightarrow [a] \Rightarrow [a] \Rightarrow [[a]]
\]
-- 1st param: input list
-- 2nd param: partial ascending sublist
\[
\text{ups2} (x:xs) [] = \text{ups2} \ x\text{s} \ [x]
\]
Accumulating parameter

Idea: Result is accumulated in parameter and returned later
Example: list of all (maximal) ascending sublists in a list
\[ \text{ups} \; [3,0,2,3,2,4] = [[3], [0,2,3], [2,4]] \]

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
\[ \text{ups} \; [3,0,2,3,2,4] = [[3], [0,2,3], [2,4]] \]

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 = [reverse (y:ys) : ups2 (x:xs) []]

How can we quickCheck the result of ups?
Accumulating parameter

The output consists of parameter and returned later
(proximal) ascending sublists in a list

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

Identifiers of list type end in ‘s’:
x, y, z, ...
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

\[ x = y + 5 \]
\[ y = x + 1 \text{ where } x = 7 \]
\[ \text{if } y = y + x \]
\[ > f 3 \]

\[ 16 \]

Binding occurrence
Scoping by example

\[ x = y + 5 \]
\[ y = x + 1 \text{ where } x = 7 \]
\[ i \ y = y + x \]

> i 3

16

**Binding occurrence**
**Bound occurrence**
**Scope of binding**

---

Scoping by example

\[ x = y + 5 \]
\[ y = x + 1 \text{ where } x = 7 \]
\[ i \ y = y + x \]

> i 3

16

**Binding occurrence**
**Bound occurrence**
**Scope of binding**
Scoping by example

\begin{align*}
x & = y + 5 \\
y & = x + 1 \text{ where } x = 7 \\
f & y = y + x \\
> f 3 \\
16
\end{align*}

Binding occurrence
Bound occurrence
Scope of binding

Summary:
- Order of definitions is irrelevant
- Parameters and where-defs are local to each equation
Scoping by example

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

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 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!
Aim

Guarantee 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

What do we prove?

Equations \( e_1 = e_2 \)
A first, simple example

Remember: \[
[] \mathbin{++} ys = ys \\
(x:xs) \mathbin{++} ys = x : (xs \mathbin{++} ys)
\]

Proof of \([1,2] \mathbin{++} [] = [1] \mathbin{++} [2]::

\[
1:2:[] \mathbin{++} [] \\
= 1 : (2:[] \mathbin{++} [])
\]
A first, simple example

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

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

\[
\begin{align*}
1:2:[[] + [] & \quad -- \text{by def of +} \\
= 1 : (2:[[] + []]) & \quad -- \text{by def of +} \\
= 1 : 2 : ([[] + []]) & \quad -- \text{by def of +} \\
= 1 : 2 : [] & \quad -- \text{by def of +} \\
= 1 : ([] + 2:[[]]) & \quad -- \text{by def of +} \\
\end{align*}
\]

A first, simple example

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

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

\[
\begin{align*}
1:2:[[] + [] & \quad -- \text{by def of +} \\
= 1 : (2:[[] + []]) & \quad -- \text{by def of +} \\
= 1 : 2 : ([[] + []]) & \quad -- \text{by def of +} \\
= 1 : 2 : [] & \quad -- \text{by def of +} \\
= 1 : ([] + 2:[[]]) & \quad -- \text{by def of +} \\
\end{align*}
\]

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),

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 \([1,2] \, ++ \, [] = [1] \, ++ \, [2]\):

\[
1:2:[] \, ++ \, [] = 1 : (2:[] \, ++ \, []) \quad -- \, by \, def \, of \, ++
\]

\[
= 1 : 2 : ([] \, ++ \, []) \quad -- \, by \, def \, of \, ++
\]

\[
= 1 : 2 : [] \quad -- \, by \, def \, of \, ++
\]

Proofs of \(e_1 = e_2\) are often better presented as two reductions to some expression \(e\):

\[
e_1 = \ldots = e
\]

\[
e_2 = \ldots = e
\]

**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.

---

**Structural induction on lists**

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 \( xs \)

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

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

\[ \uparrow \]

\textit{induction}

\textit{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 $P(x:xs)$

$\Downarrow$ induction

$\Uparrow$ new variable $x$

hypothesis ($IH$)

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** \( \text{appassoc: } (xs \ ++ \ ys) \ ++ \ zs = xs \ ++ \ (ys \ ++ \ zs) \)

**Proof** by structural induction on \( xs \)

Base case:
To show: \( (\[] \ ++ \ ys) \ ++ \ zs = \[] \ ++ \ (ys \ ++ \ zs) \)
\[
(\[] \ ++ \ ys) \ ++ \ zs \\
= ys \ ++ \ zs \quad -- \text{by def of ++}
\]
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) ++ 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) ++ 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: `((x:xs) ++ ys) ++ zs = (x:xs) ++ (ys ++ zs)``

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

`= ys ++ zs` -- by def of `++`

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

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

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

`((x:xs) ++ ys) ++ zs`
Example: associativity of ++

**Lemma** appassoc: \((xs ++ ys) ++ zs = xs ++ (ys ++ zs)\)

**Proof** by structural induction on \(xs\)

**Base case:**
To show: \(([] ++ ys) ++ zs = [] ++ (ys ++ zs)\)
\[
= \text{def of ++} = [y] ++ (ys ++ zs)
\]

**Induction step:**
IH: \(([] ++ ys) ++ zs = [] ++ (ys ++ zs)\)
To show: \(((x:xs) ++ ys) ++ zs = (x:xs) ++ (ys ++ zs)\)
\[
= \text{def of ++} = (x : (xs ++ ys)) ++ zs
\]

Example: associativity of ++

**Lemma** appassoc: \((xs ++ ys) ++ zs = xs ++ (ys ++ zs)\)

**Proof** by structural induction on \(xs\)

**Base case:**
To show: \(([] ++ ys) ++ zs = [] ++ (ys ++ zs)\)
\[
= \text{def of ++} = [y] ++ (ys ++ zs)
\]

**Induction step:**
IH: \(([] ++ ys) ++ zs = [] ++ (ys ++ zs)\)
To show: \(((x:xs) ++ ys) ++ zs = (x:xs) ++ (ys ++ zs)\)
\[
= \text{def of ++} = (x : (xs ++ ys)) ++ zs
\]

Induction template

**Lemma** \(P(xs)\)

**Proof** by structural induction on \(xs\)

**Base case:**
To show: \(P([])\)
**Lemma** $P(xs)$
**Proof** by structural induction on $xs$

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

Proof of $P([])$

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

**Example: length of ++**

**Lemma** $\text{length}(xs ++ ys) = \text{length } xs + \text{length } ys$
Example: length of ++

Lemma \( \text{length}(xs \, ++ \, ys) = \text{length} \, xs + \text{length} \, ys \)

Proof by structural induction on \(xs\)
Base case:
To show: \( \text{length} \, ([] \, ++ \, ys) = \text{length} \, [] + \text{length} \, ys \)
\( \text{length} \, ([] \, ++ \, ys) \)
\( = \text{length} \, ys \quad \text{-- by def of ++} \)
\( = 0 + \text{length} \, ys \quad \text{-- by def of length} \)

Induction step:
\( \text{IH: length}(xs \, ++ \, ys) = \text{length} \, xs + \text{length} \, ys \)

To show: \( \text{length}((x:xs)++ys) = \text{length}(x:xs) + \text{length} \, ys \)
\( \text{length}((x:xs) \, ++ \, ys) \)

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

Example: reverse of ++

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

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** \(\text{reverse}(\text{xs ++ ys}) = \text{reverse} \text{ ys} ++ \text{reverse} \text{ xs} \)

**Proof** by structural induction on \text{xs}

**Base case:**
To show: \(\text{reverse} ([] ++ \text{ys}) = \text{reverse} \text{ ys} ++ \text{reverse} []\)
- \(\text{reverse} ([] ++ \text{ys}) = \text{reverse} \text{ ys} \quad -- \text{by def of ++}\)
- \(\text{reverse} \text{ ys} ++ \text{reverse} []\)
- \(= \text{reverse} \text{ ys} ++ [] \quad -- \text{by def of reverse}\)
- \(= \text{reverse} \text{ ys} \quad -- \text{by} \)
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

**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)
reverse((x:xs) ++ ys)
= reverse(x : (xs ++ ys)) -- by def of ++
Induction step:
IH: \( \text{reverse}(xs ++ ys) = \text{reverse} ys ++ \text{reverse} xs \)
To show: \( \text{reverse}((x:xs)++ys) = \text{reverse} ys ++ \text{reverse}(x:xs) \)

\[
\begin{align*}
\text{reverse}((x:xs) ++ ys) &= \text{reverse}(x : (xs ++ ys)) & \text{-- by def of ++} \\
&= \text{reverse}(xs ++ ys) ++ [x] & \text{-- by def of reverse}
\end{align*}
\]

Induction step:
IH: \( \text{reverse}(xs ++ ys) = \text{reverse} ys ++ \text{reverse} xs \)
To show: \( \text{reverse}((x:xs)++ys) = \text{reverse} ys ++ \text{reverse}(x:xs) \)

\[
\begin{align*}
\text{reverse}((x:xs) ++ ys) &= \text{reverse}(x : (xs ++ ys)) & \text{-- by def of ++} \\
&= \text{reverse}(xs ++ ys) ++ [x] & \text{-- by def of reverse} \\
&= (\text{reverse} ys ++ \text{reverse} xs) ++ [x] & \text{-- by IH}
\end{align*}
\]

Induction step:
IH: \( \text{reverse}(xs ++ ys) = \text{reverse} ys ++ \text{reverse} xs \)
To show: \( \text{reverse}((x:xs)++ys) = \text{reverse} ys ++ \text{reverse}(x:xs) \)

\[
\begin{align*}
\text{reverse}((x:xs) ++ ys) &= \text{reverse}(x : (xs ++ ys)) & \text{-- by def of ++} \\
&= \text{reverse}(xs ++ ys) ++ [x] & \text{-- by def of reverse} \\
&= (\text{reverse} ys ++ \text{reverse} xs) ++ [x] & \text{-- by IH}
\end{align*}
\]

\[
\begin{align*}
\text{reverse} ys ++ \text{reverse}(x:xs) &= \text{reverse} ys ++ (\text{reverse} xs ++ [x]) & \text{-- by def of reverse}
\end{align*}
\]
Induction step:
IH: \(\text{reverse}(xs ++ ys) = \text{reverse} ys ++ \text{reverse} xs\)
To show: \(\text{reverse}((x:xs) ++ ys) = \text{reverse} ys ++ \text{reverse} (x:xs)\)
\[
= \text{reverse}(x : (xs ++ ys)) \quad \text{-- by def of ++}
= \text{reverse}(xs ++ ys) ++ [x] \quad \text{-- by def of reverse}
= (\text{reverse} ys ++ \text{reverse} xs) ++ [x] \quad \text{-- by IH}
= \text{reverse} ys ++ (\text{reverse} xs ++ [x]) \quad \text{-- by Lemma app.assoc}
\]
\[
= \text{reverse} ys ++ \text{reverse}(x:xs) \quad \text{-- by def of reverse}
= \text{reverse} ys ++ (\text{reverse} xs ++ [x]) \quad \text{-- by def of reverse}
\]

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

Example: reverse of ++

Lemma $\text{reverse}(xs ++ ys) = \text{reverse} ys ++ \text{reverse} xs$

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

Lemma rem z (xs ++ ys) = rem z xs ++ rem z 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
\text{Induction step:}
\text{IH: } \text{rem } z \ (x \cdot x + y \cdot y) = \text{rem } z \ x + z + \text{rem } z \ y
\text{To show: } \text{rem } z \ ((x \cdot x + y \cdot y) = \text{rem } z \ (x \cdot x) + \text{rem } z \ y
\text{Proof by cases:}
\text{Case } z = x:
\text{rem } z \ ((x \cdot x) + y \cdot y)
= \text{rem } z \ (x \cdot x + y \cdot y) \quad \text{by def of } + \text{ and rem}
= \text{rem } z \ x + z + \text{rem } z \ y \quad \text{by IH}
\\text{rem } z \ ((x \cdot x) + y \cdot y)
= \text{rem } z \ x + z + \text{rem } z \ y \quad \text{by def of rem}

\text{Induction step:}
\text{IH: } \text{rem } z \ (x \cdot x + y \cdot y) = \text{rem } z \ x + z + \text{rem } z \ y
\text{To show: } \text{rem } z \ ((x \cdot x) + y \cdot y) = \text{rem } z \ (x \cdot x) + \text{rem } z \ y
\text{Proof by cases:}
\text{Case } z = x:
\text{rem } z \ ((x \cdot x) + y \cdot y)
= \text{rem } z \ (x \cdot x + y \cdot y) \quad \text{by def of } + \text{ and rem}
= \text{rem } z \ x + z + \text{rem } z \ y \quad \text{by IH}
\text{rem } z \ ((x \cdot x) + y \cdot y)
= \text{rem } z \ x + z + \text{rem } z \ y \quad \text{by def of rem}
\text{Case } z = x:
\text{rem } z \ ((x \cdot x) + y \cdot y)
= x : \text{rem } z \ (x \cdot x + y \cdot y) \quad \text{by def of } + \text{ and rem}
= x : (\text{rem } z \ x + z + \text{rem } z \ y) \quad \text{by IH}
\text{rem } z \ ((x \cdot x) + y \cdot y)
= \text{rem } z \ x + z + \text{rem } z \ y
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

\[
\begin{align*}
\text{rem } x \ [\ ] &= \ [\ ] \\
\text{rem } x \ (y:ys) &= \text{if } x == y \text{ then rem } x \ ys \\
& \quad \text{else } y : \text{rem } x \ ys
\end{align*}
\]

Inefficiency of reverse

reverse [1,2,3]
Inefficiency of reverse

reverse [1,2,3]