PB - Übung 18.12.12

```
addTwice x y = 2 * (x+y)
```

(1) Zz addTwice x (y+z) = addTwice (x+y) z

addTwice x (y+z)

= 2*(x + (y+z)) -- Def. addTwice

= 2*((x+y) + z) -- Assoz. +

= addTwice (x+y) z -- Def. addTwice □

```
min x y = if x<y then x else y
max x y = if x<y then y else x
```

(2) Zz. min x y ≤ max x y

⇔ (if x<y then x else y) ≤ (if x<y then y else x)

- x<y (1)

⇔ x ≤ y -- if-True

⇔ True -- i.V. (1)

- x≥y (2)

⇔ y ≤ x -- if False

⇔ True -- i.V. (2)

□

(3) Zz. length (reverse xs) = length xs

- Induktion über xs
- I.A. Basis

length (reverse []) = length [] □

- I.S. Schritt

length (rev (x:xs))
= length (rev xs ++ [x]) -- Def. length
= length (rev xs) + length [x] -- Need lem!
= 1 + length (rev xs) -- Def. length, kom. +
= 1 + length xs -- Ind. annahme
= length (x:xs) -- Def. length

□

④ Zz. length (xs ++ ys) = length xs + length ys Welche Induktion?

- Induktion über xs
- Ind. basis

length ([] ++ ys)
= length ys
= 0 + length ys
= length [] + length ys □

- Ind. schritt

length (x:xs ++ ys)
= length (x : (xs ++ ys))
= 1 + length (xs ++ ys)
= (1 + length xs) + length ys -- Ind.vor, Assoz. +
= length (x:xs) + length ys

⑤ Zz. rev (rev xs) = xs

- Induktionsanfang

rev (rev []) = rev [] = [] □

- Ind. schritt

rev (rev (x:xs))
= rev (rev xs ++ [x])
= rev [x] ++ rev (rev xs) -- Needs Lemma!
= [x] ++ xs = x:xs

⑥ Zz. $rw\ (xs ++ ys) = rw\ ys ++ rw\ xs$

- Ind. basis
- $rw\ ([] ++ ys)$
- $= rw\ ys$
- $= rw\ ys ++ []$ — Neutr. lem: $a ++ [] = a$
- $= rw\ ys ++ rw\ []$ □
- Ind. schritt

$rw\ (x : xs ++ ys)$
$= rw\ (x : (xs ++ ys))$
$= rw\ (xs ++ ys) ++ [x]$
$= rw\ ys ++ (rw\ xs ++ [x])$ — Ind. annahme, Assoz. ++
$= rw\ ys ++ rw\ (x : xs)$ □

⑦ Def. `isperm :: Eq α -> [α] -> [α] -> Bool`

```
isperm xs ys = All z. cont z xs = cont z ys

cont z []     = 0
cont z (x:xs) | z == x = 1 + cont xs
              | otw.   =     cont xs
```

Zz. `xs` isperm xs (rw xs)
⇔ All z. cont z xs = cont z (rw xs)
⇔ cont z (rw xs) = cont z xs
Induktion über xs
- cont z (rw []) = cont z [] □
- cont z (rw (x : xs))
= cont z (rw xs ++ [x])
= cont z (rw xs) + cont z [x] — Neutr. Lem

cont z (xs ++ ys) = cont z xs + cont z ys

count z (rev xs) + (if z == x then 1 else 0 + 0)
= if z == x _ + count z xs   -- Ind. annahme
= count z (x : xs)

What about Shanks definition of reverse?

> foldl f z [] = z
> foldl f z (x:xs) =
>   foldl f (f z x) xs

(8) z.z. rev = foldl (flip (:)) []
⟺ rev xs = foldl (flip (:)) [] xs
Induktion über xs scheitert wg. zu schwacher Ind. voraussetzung.
Wir zeigen allgemeiner

(8a) z.z. (∀ ys) foldl (flip (:)) ys xs = rev xs ++ ys

- Basis
  foldl (flip (:)) ys [] = ys = [] ++ ys = rev [] ++ ys
- Schritt
  foldl (flip (:)) ys (x : xs)
  = foldl (flip (:)) (flip (:) ys x) xs
  = foldl (flip (:)) (x : ys) xs   -- Ind. voraussetzung! Wichtig: ∀ ys.
  = rev xs ++ x : ys
  = (rev xs ++ [x]) ++ ys
  = rev (x : xs) ++ ys. □

Damit folgt (8) direkt mit ys = []. □

⑨ Einfaches Beispiel Bäume

```
flipT :: Tree α → Tree α
flipT Null = Null
flipT (Node l v r) = Node (flipT r) v (flipT l)
```

22. `flipT (flipT t) = t`

- Ind. Anfang

```
  flipT (flipT Null)
= flipT Null
= Null                    □
```

- Ind. Schritt

```
  flipT (flipT (Node l v r))
= flipT (Node (flipT r) v (flipT l))          -Def. flip
= flipNode (flipT (flipT l)) v (flipT (flipT r))   -Ind. Anf.
= Node l v r                                  □
```

(10) Beweis der Eigenschaften des Datentyps Map

- Beweis über Tree selbst schwierig, unnötig kompliziert wegen vieler Fälle. Einfacher: Operation von Tree auf List zurückführen:

```
toList :: Tree α → [α]
```

mit zwei Repräsentationstheoremen

(A) `lookup k t = List.lookup k (toList t)`

(B) `toList (put (k, v) t) = insUpd k v (toList t)`

```
insUpd k Nothing xs = remove k xs
insUpd k (Just v) xs = insert (k,v) xs

remove k [] = []
remove k ((m,s):xs) | k == m = remove k xs
                    | otherwise = (m,s) : remove k xs

insert (k,v) [] = [(k,v)]
insert (k,v) ((m,s):xs) | k < m = (k,v):(m,s):xs
                        | k == m = (k,v):xs
                        | k > m = (m,s) : insert (k,v) xs
```

insUpd

insUpd (l, NoMsg) xs = remove l xs

insUpd (l, Just v) xs = insert (l,v) xs

remove l [] = []

remove l ((m,s) : xs) | l == m = xs remove l xs

| l /= m = (m,s) : remove l xs

insert (l,v) [] = [(l,v)]

insert (l,v) (m,s) : xs | l < m = (l,v) : (m,s) : xs

| l == m = (l,v) : xs

| l > m = (m,s) : insert (l,v) xs

Damit

lookup l (insUpd (l, NoMsg) xs) = NoMsg

— Ind. Schritt:

lookup l (remove l ((m,s) : xs)

1. l == m

lookup l (remove l xs) ≠ NoMsg ↯ — Ind. annahme

2. l /= m

lookup l ((m,s) : remove l xs)

= lookup l (remove l xs) ← NoMsg

= NoMsg

lookup l (insert (l,v) xs) = just v

-- Ind. Schritt

lookup l (insert (l,v) ((m,s):xs)) =

1. l < m

= lookup l ((l,v):(m,s):xs))

= just w □

2. l = m

= lookup l ((l,v):xs)

= just w □

3. l > m

= lookup l ((m,s): insert (l,v) xs))

= lookup l (insert (l,v) xs)

= just w □

Vergleichbar:

l /= m ⇒ lookup l (insert (m,v) xs) = lookup l xs

~~Ind. Annahme~~

~~lookup l (insert (m,v) xs) = if l = m then v else insert lookup l xs~~ No.

l ≠ m ⇒ lookup l (remove m xs) = lookup l xs

-Ind. Schritt

lookup l (remove m ((n,t):xs)

1. m = n, l ≠ n

= lookup l (remove m xs) = lookup l xs Ind.-Vor.

= lookup l ((n,t):xs) l ≠ n

2. $m \neq n$, $l \geq n$

= lookup l ((n,t) : remove m xs)

= Just t

= lookup l ((n,t) : xs) -- $l \geq n$

3. $m \neq n$, $l \neq n$

= lookup l ((n,t) : remove m xs)

= lookup l (remove m xs)

= lookup l xs -- Ind. annahme

= lookup l ((n,t) : xs) -- $n \neq l$

- Analoges für insert, mit mehr Fällen...