Strukturelle Induktion über Peano-Zahlen — Beispiel

```
data Nat = Z | S Nat
double :: Nat -> Nat
double x = case x of
  Z    -> Z
  S x' ->  S ( S ( double x' ) )
plus :: Nat -> Nat -> Nat
plus x y     = case x of
  Z    -> y
  S x' ->  S ( plus x' y )
```

Strukturelle Induktion zum Nachweis von

```
double x .=. plus x x
```

IA: `x = Z`

```
                              double Z
        (by def double)  .=.  Z
        (by def plus)    .=.  plus Z Z
```

IS: `x = S x'`

IH: `double x' .=. plus x' x'`

z.z.: `double (S x') .=. plus (S x') (S x')`

Wiederholung – Datentyp Liste (polymorph)

```
data List a = Nil
    | Cons { head :: a, tail :: List a}
```

vordefiniert:

```
data [a] = [] | a : [a]
```

Pattern Matching:

```
f :: [a] -> ...
f xs = case xs of
    []         -> ...
    (x : xss) -> ...
```

Beispiel:

```
app :: [a] -> [a] -> [a]
app xs ys = case xs of
    []         -> ys
    (x : xss) -> x : (app xss ys)
```

Strukturelle Induktion über Listen – Beispiel

zum Nachweis von Eigenschaften wie z.B.

- `app xs [] = xs`
- `app` ist assoziativ, d.h

```
app xs (app ys zs) = app (app xs ys) zs
```

Länge der Eingabeliste

```
len :: [a] -> Int
len xs = case xs of
    []         -> 0
    (x : xss) -> 1 + len xss
```

Strukturelle Induktion zum Nachweis von
ÜA: `len ( app xs ys ) = len xs + len ys`

Strukturelle Induktion über Listen – Beispiel

```
data Nat = Z | S Nat
data List a = Nil | Cons a (List a)
doubles :: List Nat -> List Nat
doubles xs = case xs of
  Nil         -> Nil
  Cons x xss -> Cons (double x) ( doubles xss )
sum :: List Nat -> Nat
sum xs = case xs of
  Nil         -> Z
  Cons x xss -> plus x ( sum xss )
```

Strukturelle Induktion zum Nachweis von

`sum ( doubles xs ) .=. double ( sum xs )`

IA: `xs = Nil`

`sum ( doubles Nil ) .=. ...`

IS: `xs = Cons x xss`

IH: `sum (doubles xss) .=. double (sum xss)`

z.z.: `sum ( doubles (Cons x xss) ) .=. ...`

Datentyp Binärbaum (polymorph)

```
data Bintree a = Leaf {}
      | Branch { left  :: Bintree a,
                 key   :: a,
                 right :: Bintree a }
```

Beispiel:

```
t :: Bintree Int
t = Branch {
    left = Branch { left = Leaf {},
                    key = 5,
                    right = Leaf {} },
    key = 3,
    right = Branch {
              left = Leaf {},
              key = 2,
              right = Branch { left = Leaf {},
                               key = 4,
                               right = Leaf {} }}}
```

Pattern Matching

```
data Bintree a = Leaf {}
      | Branch { left  :: Bintree a,
                 key   :: a,
                 right :: Bintree a }


f :: Bintree a -> ..
f t = case t of
    Leaf {} -> ..
    Branch {} -> ..
```

oder tiefer:

```
f :: Bintree a -> ..
f t = case t of
    Leaf {} -> ..
    Branch { left = l, key = k, right = r } -> ..
```

Rekursion über binäre Bäume – Beispiele

Anzahl der inneren Knoten

```
size :: Bintree a -> Int
size t = case t of
    Leaf {}   -> 0
    Branch {} -> size (left t)
        + 1 + size (right t)
```

Anzahl der Blätter:

```
leaves :: Bintree a -> Int
leaves t = case t of
    Leaf   {} -> ...
    Branch {} -> ...
```

Summe der Schlüssel (Int):

```
bt_sum :: Bintree Int -> Int
bt_sum t = case t of
    Leaf   {} -> ...
    Branch {} -> ...
```

Mehr Beispiele

jeden Schlüssel verdoppeln

```
doubles :: Bintree Int -> Bintree Int
doubles t = case t of
    Leaf {}   -> Leaf {}
    Branch {} -> ...

inorder :: Bintree a -> [a]
inorder t = case t of
    Leaf {} -> []
    Branch {} -> ...
```

vollständiger binärer Baum der Höhe `h`:

```
full :: Int -> Bintree Int
full h = if h > 0
    then Branch { left = full (h-1),
                  key = h,
                  right = full (h-1) }
    else Leaf {}
```

Strukturelle Induktion über Binärbäume

z.z. Jeder Binärbaum `t` mit Schlüsseln vom Typ `a`
hat die Eigenschaft `P`

IA (`t = Leaf`): z.z.: `Leaf` hat die Eigenschaft `P`

IS IV: Binärbäume `l` und `r` erfüllen `P`

IB: $\forall$ `k :: a` hat der Binärbaum

```
Branch { left = l,
         key = k,
         right = r }
```

die Eigenschaft `P`

zum Nachweis von Eigenschaften wie z.B.

- $\forall$ `( t :: Bintree Nat ) :`
  `bt_sum (doubles t) = double ( bt_sum t )`
- $\forall$ `( t :: Bintree Int ) :`
  `bt_sum t = sum ( inorder t )`