Beispielbeweise zur Vorlesung vom 17.12.2008

Christoph Lüth

4. Januar 2009

Wir nehmen als folgende Gleichungen für `length`, `++`, `filter`, `map` folgende an:

```
length [] = 0
length (x:xs) = 1+ length xs

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

filter p [] = []
filter p (x:xs) = if p x then x: filter p xs
                         else filter p xs

map f []     = []
map f (x:xs) = f x: map f xs
```

Dies müssen nicht unbedingt die definieren Gleichungen sein (`map` könnte beispielsweise auch als

```
map f = foldr ((:). f) []
```

definiert sein), aber die Gleichungen müssen gelten.

zz: `length (filter p xs) <= length xs`

Ind: 1. Induktionsbasis

`length (filter p [])`

$=$ `length []` Def. `filter`

2. Induktionsschritt

`length (filter p (x:xs))`

2.1. `p x`

$=$	`length (x:filter p xs)`	Def. `filter`
$=$	`1+ length (filter p xs)`	Def. `length`
$\leq$	`1+ length xs`	Ind.vor.
$=$	`length (x:xs)`	Def. `length`

2.2 $\neg$`p x`

$=$	`length (filter p xs)`	Def. `filter`
$\leq$	`length xs`	Ind.vor.
$\leq$	`1+ length xs`	$x \leq x + 1$
$\leq$	`length (x:xs)`	Def. `length`

□

zz: `length (xs++ ys) == length xs+ length ys`

Induktion über `xs`

1. Induktionbasis

	`length ([]++ ys)`	
=	`length ys`	Def. `++`
=	`0+ length ys`	$0 + x = x$
=	`length []+ length ys`	Def. `length`

2. Induktionsschritt

	`length (x:xs ++ ys)`	
=	`length (x:(xs++ ys))`	Def. `++`
=	`1+ length (xs ++ ys)`	Def. `length`
=	`1+ (length xs+ length ys)`	Ind.vor.
=	`(1+length xs)+ length ys`	Assoziativität von `+`
=	`length (x:xs)+ length ys`	Def. `length`

□

zz: `map f (xs++ ys) == map f xs++ map f ys`

Induktion über `xs`

1. Induktionsbasis

	`map f ([]++ ys`	
=	`map f ys`	Def. `++`
=	`[]++ map f ys`	Def. `++`
=	`map f [] ++ map f ys`	Def. `map`

2. Induktionsschritt

	`map f (x:xs ++ ys)`	
=	`map f (x:(xs++ ys))`	Def. `++`
=	`f x: map f (xs++ ys)`	Def. `map`
=	`f x: (map f xs ++ map f ys)`	Ind.vor.
=	`(f x: map f xs) ++ map f ys`	Def. `++`
=	`map f (x:xs) ++ map f ys`	Def. `map`

□

Wir nehmen ferner folgende Gleichungen für `sum` und `concat` an:

```
sum :: [Int]-> Int
sum []     = 0
sum (x:xs) = x+ sum xs

concat :: [[a]]-> [a]
concat []       = []
concat (xs:xss) = xs++ concat xs
```

zz: `sum (map length xs)== length (concat xs)`

Ind: 1. Induktionsschritt

	`sum (map length [])`	
=	`sum []`	Def. `map`
=	`0`	Def. `length`
=	`length []`	Def. `length`
=	`length (concat [])`	Def. `concat`

2. Induktionsschritt

	`sum (map length (x:xs))`	
=	`sum (length x: map length xs)`	Def. `map`
=	`length x+ sum (map length xs)`	Def. `sum`
=	`length x+ length (concat xs)`	Ind.vor.
=	`length (x++ concat xs)`	Lemma (2) oben
=	`length (concat (x:xs))`	Def. `concat`

□