

Eine Behauptung ...

Schon bekannte Funktionen:

```
filter :: (a → Bool) → [a] → [a]
```

```
filter p [] = []
```

```
filter p (a : as) | p a = a : (filter p as)  
                  | otherwise = filter p as
```

```
map :: (a → b) → [a] → [b]
```

```
map h [] = []
```

```
map h (a : as) = (h a) : (map h as)
```

Eine Behauptung ...

Schon bekannte Funktionen:

```
filter :: (a -> Bool) -> [a] -> [a]
filter p []      = []
filter p (a : as) | p a      = a : (filter p as)
                  | otherwise = filter p as
```

```
map :: (a -> b) -> [a] -> [b]
map h []      = []
map h (a : as) = (h a) : (map h as)
```

Für jede Wahl von p , h und l gilt:

```
filter p (map h l) =
```

Eine Behauptung ...

Schon bekannte Funktionen:

```
filter :: (a → Bool) → [a] → [a]
filter p []      = []
filter p (a : as) | p a      = a : (filter p as)
                  | otherwise = filter p as
```

```
map :: (a → b) → [a] → [b]
map h []      = []
map h (a : as) = (h a) : (map h as)
```

Für jede Wahl von p , h und l gilt:

```
filter p (map h l) = map h (filter (p ∘ h) l)
```

... und ihr Beweis

Per Induktion über die Länge der Liste:

1. `filter p (map h []) = map h (filter (p ∘ h) [])` ?

... und ihr Beweis

Per Induktion über die Länge der Liste:

1. `filter p (map h []) = map h (filter (p ∘ h) [])` ? OK

... und ihr Beweis

Per Induktion über die Länge der Liste:

1. `filter p (map h []) = map h (filter (p ∘ h) [])` ? OK
2. `filter p (map h (a : as)) = map h (filter (p ∘ h) (a : as))` ?

... und ihr Beweis

Per Induktion über die Länge der Liste:

1. `filter p (map h [])` = `map h (filter (p ∘ h) [])` ? OK
2. `filter p (map h (a : as))` = `map h (filter (p ∘ h) (a : as))` ?
`filter p (map h (a : as))`

... und ihr Beweis

Per Induktion über die Länge der Liste:

1. `filter p (map h []) = map h (filter (p ∘ h) [])` ? OK
2. `filter p (map h (a : as)) = map h (filter (p ∘ h) (a : as))` ?
$$\begin{aligned} & \text{filter } p (\text{map } h (a : as)) \\ &= \text{filter } p ((h a) : (\text{map } h as)) \end{aligned}$$

... und ihr Beweis

Per Induktion über die Länge der Liste:

1. `filter p (map h []) = map h (filter (p ∘ h) [])` ? OK
2. `filter p (map h (a : as)) = map h (filter (p ∘ h) (a : as))` ?

$$\begin{aligned} & \text{filter } p (\text{map } h (a : as)) \\ = & \text{filter } p ((h a) : (\text{map } h as)) \\ = & \text{if } p (h a) \text{ then } (h a) : (\text{filter } p (\text{map } h as)) \\ & \quad \text{else } \text{filter } p (\text{map } h as) \end{aligned}$$

... und ihr Beweis

Per Induktion über die Länge der Liste:

1. `filter p (map h []) = map h (filter (p ∘ h) [])` ? OK
2. `filter p (map h (a : as)) = map h (filter (p ∘ h) (a : as))` ?

$$\begin{aligned} & \text{filter } p (\text{map } h (a : as)) \\ = & \text{filter } p ((h a) : (\text{map } h as)) \\ = & \text{if } p (h a) \text{ then } (h a) : (\text{filter } p (\text{map } h as)) \\ & \quad \text{else } \text{filter } p (\text{map } h as) \\ = & \text{if } (p \circ h) a \text{ then } (h a) : (\text{map } h (\text{filter } (p \circ h) as)) \\ & \quad \text{else } \text{map } h (\text{filter } (p \circ h) as) \end{aligned}$$

... und ihr Beweis

Per Induktion über die Länge der Liste:

1. `filter p (map h []) = map h (filter (p ∘ h) [])` ? OK
2. `filter p (map h (a : as)) = map h (filter (p ∘ h) (a : as))` ?

$$\begin{aligned} & \text{filter } p (\text{map } h (a : as)) \\ = & \text{filter } p ((h a) : (\text{map } h as)) \\ = & \text{if } p (h a) \text{ then } (h a) : (\text{filter } p (\text{map } h as)) \\ & \quad \text{else } \text{filter } p (\text{map } h as) \\ = & \text{if } (p \circ h) a \text{ then } (h a) : (\text{map } h (\text{filter } (p \circ h) as)) \\ & \quad \text{else } \text{map } h (\text{filter } (p \circ h) as) \\ = & \text{if } (p \circ h) a \text{ then } \text{map } h (a : (\text{filter } (p \circ h) as)) \\ & \quad \text{else } \text{map } h (\text{filter } (p \circ h) as) \end{aligned}$$

... und ihr Beweis

Per Induktion über die Länge der Liste:

1. `filter p (map h []) = map h (filter (p ∘ h) [])` ? OK
2. `filter p (map h (a : as)) = map h (filter (p ∘ h) (a : as))` ?

$$\begin{aligned} & \text{filter } p (\text{map } h (a : as)) \\ = & \text{filter } p ((h a) : (\text{map } h as)) \\ = & \text{if } p (h a) \text{ then } (h a) : (\text{filter } p (\text{map } h as)) \\ & \quad \text{else } \text{filter } p (\text{map } h as) \\ = & \text{if } (p \circ h) a \text{ then } (h a) : (\text{map } h (\text{filter } (p \circ h) as)) \\ & \quad \text{else } \text{map } h (\text{filter } (p \circ h) as) \\ = & \text{if } (p \circ h) a \text{ then } \text{map } h (a : (\text{filter } (p \circ h) as)) \\ & \quad \text{else } \text{map } h (\text{filter } (p \circ h) as) \\ = & \text{map } h (\text{if } (p \circ h) a \text{ then } a : (\text{filter } (p \circ h) as) \\ & \quad \text{else } \text{filter } (p \circ h) as) \end{aligned}$$

... und ihr Beweis

Per Induktion über die Länge der Liste:

1. `filter p (map h [])` = `map h (filter (p ∘ h) [])` ? OK
2. `filter p (map h (a : as))` = `map h (filter (p ∘ h) (a : as))` ?

$$\begin{aligned} & \text{filter } p (\text{map } h (a : as)) \\ = & \text{filter } p ((h a) : (\text{map } h as)) \\ = & \text{if } p (h a) \text{ then } (h a) : (\text{filter } p (\text{map } h as)) \\ & \quad \text{else } \text{filter } p (\text{map } h as) \\ = & \text{if } (p \circ h) a \text{ then } (h a) : (\text{map } h (\text{filter } (p \circ h) as)) \\ & \quad \text{else } \text{map } h (\text{filter } (p \circ h) as) \\ = & \text{if } (p \circ h) a \text{ then } \text{map } h (a : (\text{filter } (p \circ h) as)) \\ & \quad \text{else } \text{map } h (\text{filter } (p \circ h) as) \\ = & \text{map } h (\text{if } (p \circ h) a \text{ then } a : (\text{filter } (p \circ h) as) \\ & \quad \quad \quad \text{else } \text{filter } (p \circ h) as) \\ = & \text{map } h (\text{filter } (p \circ h) (a : as)) \end{aligned}$$

... und ihr Beweis

Per Induktion über die Länge der Liste:

1. `filter p (map h []) = map h (filter (p ∘ h) [])` ? OK
2. `filter p (map h (a : as)) = map h (filter (p ∘ h) (a : as))` ?

$$\begin{aligned} & \text{filter } p (\text{map } h (a : as)) \\ = & \text{filter } p ((h a) : (\text{map } h as)) \\ = & \text{if } p (h a) \text{ then } (h a) : (\text{filter } p (\text{map } h as)) \\ & \quad \text{else } \text{filter } p (\text{map } h as) \\ = & \text{if } (p \circ h) a \text{ then } (h a) : (\text{map } h (\text{filter } (p \circ h) as)) \\ & \quad \text{else } \text{map } h (\text{filter } (p \circ h) as) \\ = & \text{if } (p \circ h) a \text{ then } \text{map } h (a : (\text{filter } (p \circ h) as)) \\ & \quad \text{else } \text{map } h (\text{filter } (p \circ h) as) \\ = & \text{map } h (\text{if } (p \circ h) a \text{ then } a : (\text{filter } (p \circ h) as) \\ & \quad \quad \quad \text{else } \text{filter } (p \circ h) as) \\ = & \text{map } h (\text{filter } (p \circ h) (a : as)) \end{aligned}$$

OK, sofern

$$\text{filter } p (\text{map } h as) = \text{map } h (\text{filter } (p \circ h) as)$$

Ein weiteres Beispiel

Angenommen, wir wollen (für gewisse h , f , k und l)

$$h (\text{foldr } f \ k \ l)$$

durch einen einzelnen Aufruf von `foldr` ersetzen.

Ein weiteres Beispiel

Angenommen, wir wollen (für gewisse h , f , k und l)

$$h (\text{foldr } f \ k \ l)$$

durch einen einzelnen Aufruf von `foldr` ersetzen.

The diagram shows the transformation of a nested foldr expression into a single foldr call. On the left, the expression $h (\text{foldr } f \ k (\dots))$ is shown with a tree structure. The inner foldr has a root node \cdot with children x_1 and \cdot . This second \cdot has children x_2 and \dots . The final \cdot has children x_n and $[]$. On the right, the equivalent single foldr call is shown with a tree structure where the root node is h , which has a single child f . This f node has children x_1 and f . The second f node has children x_2 and \dots . The final f node has children x_n and k . The two trees are separated by an equals sign.

Ein weiteres Beispiel

Angenommen, wir wollen (für gewisse h , f , k und l)

$$h (\text{foldr } f \ k \ l)$$

durch einen einzelnen Aufruf von `foldr` ersetzen.

The diagram illustrates the equivalence between a nested foldr expression and a single foldr call. On the left, the expression $h (\text{foldr } f \ k \ l)$ is shown as a tree structure. The root node is h , which has a single child node f . This f node has two children: x_1 and another f node. This second f node has children x_2 and a third f node. This pattern continues until the final f node has children x_n and an empty list $[]$. On the right, the expression $h (\text{foldr } f \ k \ l)$ is shown as a tree structure. The root node is h , which has a single child node f . This f node has two children: x_1 and another f node. This second f node has children x_2 and a third f node. This pattern continues until the final f node has children x_n and k . The two trees are separated by an equals sign, indicating their equivalence.

Angenommen, wir wüssten ein g , so dass für alle x und y ,

$$h (f \ x \ y) = g \ x \ (h \ y)$$

..., dann:

$$h (\text{foldr } f \ k \ l) = \text{foldr } g \ (h \ k) \ l$$

..., dann:

$$h (\text{foldr } f \ k \ l) = \text{foldr } g \ (h \ k) \ l$$

Beweis per Induktion über die Länge der Liste:

1. $h (\text{foldr } f \ k \ []) = \text{foldr } g \ (h \ k) \ []$?

..., dann:

$$h (\text{foldr } f \ k \ l) = \text{foldr } g \ (h \ k) \ l$$

Beweis per Induktion über die Länge der Liste:

1. $h (\text{foldr } f \ k \ []) = \text{foldr } g \ (h \ k) \ []$? OK

..., dann:

$$h (\text{foldr } f \ k \ l) = \text{foldr } g \ (h \ k) \ l$$

Beweis per Induktion über die Länge der Liste:

1. $h (\text{foldr } f \ k \ []) = \text{foldr } g \ (h \ k) \ []$? OK
2. $h (\text{foldr } f \ k \ (a : as)) = \text{foldr } g \ (h \ k) \ (a : as)$?

..., dann:

$$h (\text{foldr } f \ k \ l) = \text{foldr } g \ (h \ k) \ l$$

Beweis per Induktion über die Länge der Liste:

1. $h (\text{foldr } f \ k \ []) = \text{foldr } g \ (h \ k) \ []$? OK
2. $h (\text{foldr } f \ k \ (a : as)) = \text{foldr } g \ (h \ k) \ (a : as)$?

$$h (\text{foldr } f \ k \ (a : as))$$

..., dann:

$$h (\text{foldr } f \ k \ l) = \text{foldr } g \ (h \ k) \ l$$

Beweis per Induktion über die Länge der Liste:

1. $h (\text{foldr } f \ k \ []) = \text{foldr } g \ (h \ k) \ [] \ ? \text{ OK}$
2. $h (\text{foldr } f \ k \ (a : as)) = \text{foldr } g \ (h \ k) \ (a : as) \ ?$

$$\begin{aligned} & h (\text{foldr } f \ k \ (a : as)) \\ = & h (f \ a \ (\text{foldr } f \ k \ as)) \end{aligned}$$

..., dann:

$$h (\text{foldr } f \ k \ l) = \text{foldr } g \ (h \ k) \ l$$

Beweis per Induktion über die Länge der Liste:

1. $h (\text{foldr } f \ k \ []) = \text{foldr } g \ (h \ k) \ [] \ ? \text{ OK}$
2. $h (\text{foldr } f \ k \ (a : as)) = \text{foldr } g \ (h \ k) \ (a : as) \ ?$

$$\begin{aligned} & h (\text{foldr } f \ k \ (a : as)) \\ = & h (f \ a \ (\text{foldr } f \ k \ as)) \\ = & g \ a \ (h (\text{foldr } f \ k \ as)) \end{aligned}$$

..., dann:

$$h (\text{foldr } f \ k \ l) = \text{foldr } g \ (h \ k) \ l$$

Beweis per Induktion über die Länge der Liste:

1. $h (\text{foldr } f \ k \ []) = \text{foldr } g \ (h \ k) \ [] \ ? \text{ OK}$
2. $h (\text{foldr } f \ k \ (a : as)) = \text{foldr } g \ (h \ k) \ (a : as) \ ?$

$$\begin{aligned} & h (\text{foldr } f \ k \ (a : as)) \\ = & h (f \ a \ (\text{foldr } f \ k \ as)) \\ = & g \ a \ (h (\text{foldr } f \ k \ as)) \\ = & g \ a \ (\text{foldr } g \ (h \ k) \ as) \end{aligned}$$

..., dann:

$$h (\text{foldr } f \ k \ l) = \text{foldr } g \ (h \ k) \ l$$

Beweis per Induktion über die Länge der Liste:

1. $h (\text{foldr } f \ k \ []) = \text{foldr } g \ (h \ k) \ [] \ ? \text{ OK}$
2. $h (\text{foldr } f \ k \ (a : as)) = \text{foldr } g \ (h \ k) \ (a : as) \ ?$

$$\begin{aligned} & h (\text{foldr } f \ k \ (a : as)) \\ = & h (f \ a \ (\text{foldr } f \ k \ as)) \\ = & g \ a \ (h (\text{foldr } f \ k \ as)) \\ = & g \ a \ (\text{foldr } g \ (h \ k) \ as) \\ = & \text{foldr } g \ (h \ k) \ (a : as) \end{aligned}$$

..., dann:

$$h (\text{foldr } f \ k \ l) = \text{foldr } g \ (h \ k) \ l$$

Beweis per Induktion über die Länge der Liste:

1. $h (\text{foldr } f \ k \ []) = \text{foldr } g \ (h \ k) \ []$? OK
2. $h (\text{foldr } f \ k \ (a : as)) = \text{foldr } g \ (h \ k) \ (a : as)$? OK

Auf anderen Datentypen

Zum Beispiel:

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
```

Auf anderen Datentypen

Zum Beispiel:

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
```

```
flat :: Tree a → [a]
```

```
flat (Leaf a)      = [a]
```

```
flat (Node t1 t2) = (flat t1) ++ (flat t2)
```

Auf anderen Datentypen

Zum Beispiel:

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
```

```
flat :: Tree a → [a]
```

```
flat (Leaf a)      = [a]
```

```
flat (Node t1 t2) = (flat t1) ++ (flat t2)
```

```
incr :: Tree Int → Tree Int
```

```
incr (Leaf a)      = Leaf (a + 1)
```

```
incr (Node t1 t2) = Node (incr t1) (incr t2)
```

Auf anderen Datentypen

Zum Beispiel:

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
```

```
flat :: Tree a → [a]
```

```
flat (Leaf a)      = [a]
```

```
flat (Node t1 t2) = (flat t1) ++ (flat t2)
```

```
incr :: Tree Int → Tree Int
```

```
incr (Leaf a)      = Leaf (a + 1)
```

```
incr (Node t1 t2) = Node (incr t1) (incr t2)
```

Behauptung:

```
flat (incr t) =
```

Auf anderen Datentypen

Zum Beispiel:

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
```

```
flat :: Tree a → [a]
```

```
flat (Leaf a)    = [a]
```

```
flat (Node t1 t2) = (flat t1) ++ (flat t2)
```

```
incr :: Tree Int → Tree Int
```

```
incr (Leaf a)    = Leaf (a + 1)
```

```
incr (Node t1 t2) = Node (incr t1) (incr t2)
```

Behauptung:

```
flat (incr t) = map (+1) (flat t)
```


Auf anderen Datentypen

Beweis per struktureller Induktion:

1. `flat (incr (Leaf a)) = map (+1) (flat (Leaf a)) ?`

Auf anderen Datentypen

Beweis per struktureller Induktion:

1. `flat (incr (Leaf a)) = map (+1) (flat (Leaf a)) ?`
OK

Auf anderen Datentypen

Beweis per struktureller Induktion:

1. $\text{flat} (\text{incr} (\text{Leaf } a)) = \text{map} (+1) (\text{flat} (\text{Leaf } a)) ?$

OK

2. $\text{flat} (\text{incr} (\text{Node } t_1 t_2)) = \text{map} (+1) (\text{flat} (\text{Node } t_1 t_2)) ?$

Auf anderen Datentypen

Beweis per struktureller Induktion:

1. `flat (incr (Leaf a)) = map (+1) (flat (Leaf a)) ?`

OK

2. `flat (incr (Node t1 t2)) = map (+1) (flat (Node t1 t2)) ?`

`flat (incr (Node t1 t2))`

Auf anderen Datentypen

Beweis per struktureller Induktion:

1. `flat (incr (Leaf a)) = map (+1) (flat (Leaf a)) ?`

OK

2. `flat (incr (Node t1 t2)) = map (+1) (flat (Node t1 t2)) ?`

$$\begin{aligned} & \text{flat (incr (Node } t_1 \ t_2)) \\ = & \text{flat (Node (incr } t_1) \ (\text{incr } t_2)) \end{aligned}$$

Auf anderen Datentypen

Beweis per struktureller Induktion:

1. `flat (incr (Leaf a)) = map (+1) (flat (Leaf a)) ?`
OK

2. `flat (incr (Node t1 t2)) = map (+1) (flat (Node t1 t2)) ?`

$$\begin{aligned} & \text{flat (incr (Node } t_1 \ t_2)) \\ = & \text{flat (Node (incr } t_1) \ (\text{incr } t_2)) \\ = & (\text{flat (incr } t_1)) \ ++ \ (\text{flat (incr } t_2)) \end{aligned}$$

Auf anderen Datentypen

Beweis per struktureller Induktion:

1. `flat (incr (Leaf a)) = map (+1) (flat (Leaf a)) ?`

OK

2. `flat (incr (Node t1 t2)) = map (+1) (flat (Node t1 t2)) ?`

$$\begin{aligned} & \text{flat (incr (Node } t_1 \ t_2)) \\ = & \text{flat (Node (incr } t_1) \ (\text{incr } t_2)) \\ = & (\text{flat (incr } t_1)) \ ++ \ (\text{flat (incr } t_2)) \\ = & (\text{map (+1) (flat } t_1)) \ ++ \ (\text{map (+1) (flat } t_2)) \end{aligned}$$

Auf anderen Datentypen

Beweis per struktureller Induktion:

1. `flat (incr (Leaf a)) = map (+1) (flat (Leaf a)) ?`
OK

2. `flat (incr (Node t1 t2)) = map (+1) (flat (Node t1 t2)) ?`

$$\begin{aligned} & \text{flat (incr (Node } t_1 \ t_2)) \\ = & \text{flat (Node (incr } t_1) \ (\text{incr } t_2)) \\ = & (\text{flat (incr } t_1)) \ ++ \ (\text{flat (incr } t_2)) \\ = & (\text{map (+1) (flat } t_1)) \ ++ \ (\text{map (+1) (flat } t_2)) \\ = & \text{map (+1) ((flat } t_1) \ ++ \ (\text{flat } t_2)) \end{aligned}$$

Auf anderen Datentypen

Beweis per struktureller Induktion:

1. `flat (incr (Leaf a)) = map (+1) (flat (Leaf a)) ?`

OK

2. `flat (incr (Node t1 t2)) = map (+1) (flat (Node t1 t2)) ?`

$$\begin{aligned} & \text{flat (incr (Node } t_1 \ t_2)) \\ = & \text{flat (Node (incr } t_1) \ (\text{incr } t_2)) \\ = & (\text{flat (incr } t_1)) \ ++ \ (\text{flat (incr } t_2)) \\ = & (\text{map (+1) (flat } t_1)) \ ++ \ (\text{map (+1) (flat } t_2)) \\ = & \text{map (+1) ((flat } t_1) \ ++ \ (\text{flat } t_2)) \\ = & \text{map (+1) (flat (Node } t_1 \ t_2)) \end{aligned}$$

Auf anderen Datentypen

Beweis per struktureller Induktion:

1. $\text{flat} (\text{incr} (\text{Leaf } a)) = \text{map} (+1) (\text{flat} (\text{Leaf } a))$?
OK

2. $\text{flat} (\text{incr} (\text{Node } t_1 t_2)) = \text{map} (+1) (\text{flat} (\text{Node } t_1 t_2))$?
OK (mit Hilfe: $(\text{map } h l_1) ++ (\text{map } h l_2) = \text{map } h (l_1 ++ l_2)$)

Manchmal nicht so einfach

Naives Umdrehen einer Liste:

`reverse` :: $[a] \rightarrow [a]$

`reverse` [] = []

`reverse` (a : as) = (`reverse` as) ++ [a]

Manchmal nicht so einfach

Naives Umdrehen einer Liste:

`reverse` :: $[a] \rightarrow [a]$

`reverse` [] = []

`reverse` (a : as) = (`reverse` as) ++ [a]

Zu beweisen:

`map` h (`reverse` l) = `reverse` (`map` h l)

Manchmal nicht so einfach

Naives Umdrehen einer Liste:

```
reverse :: [a] → [a]
reverse []      = []
reverse (a : as) = (reverse as) ++ [a]
```

Zu beweisen:

$$\text{map } h (\text{reverse } l) = \text{reverse } (\text{map } h l)$$

Induktionsschritt:

2. $\text{map } h (\text{reverse } (a : as)) = \text{reverse } (\text{map } h (a : as)) ?$

Manchmal nicht so einfach

Naives Umdrehen einer Liste:

```
reverse :: [a] → [a]
reverse []      = []
reverse (a : as) = (reverse as) ++ [a]
```

Zu beweisen:

$$\text{map } h (\text{reverse } l) = \text{reverse } (\text{map } h l)$$

Induktionsschritt:

2. $\text{map } h (\text{reverse } (a : as)) = \text{reverse } (\text{map } h (a : as)) ?$

$$\text{map } h (\text{reverse } (a : as))$$

Manchmal nicht so einfach

Naives Umdrehen einer Liste:

```
reverse :: [a] → [a]
reverse []      = []
reverse (a : as) = (reverse as) ++ [a]
```

Zu beweisen:

$$\text{map } h (\text{reverse } l) = \text{reverse } (\text{map } h l)$$

Induktionsschritt:

$$2. \text{map } h (\text{reverse } (a : as)) = \text{reverse } (\text{map } h (a : as)) ?$$

$$\begin{aligned} & \text{map } h (\text{reverse } (a : as)) \\ &= \text{map } h ((\text{reverse } as) ++ [a]) \end{aligned}$$

Manchmal nicht so einfach

Naives Umdrehen einer Liste:

```
reverse :: [a] → [a]
reverse []      = []
reverse (a : as) = (reverse as) ++ [a]
```

Zu beweisen:

$$\text{map } h (\text{reverse } l) = \text{reverse } (\text{map } h l)$$

Induktionsschritt:

$$2. \text{map } h (\text{reverse } (a : as)) = \text{reverse } (\text{map } h (a : as)) ?$$

$$\begin{aligned} & \text{map } h (\text{reverse } (a : as)) \\ &= \text{map } h ((\text{reverse } as) ++ [a]) \\ &= (\text{map } h (\text{reverse } as)) ++ (\text{map } h [a]) \end{aligned}$$

Manchmal nicht so einfach

Naives Umdrehen einer Liste:

```
reverse :: [a] → [a]
reverse []      = []
reverse (a : as) = (reverse as) ++ [a]
```

Zu beweisen:

```
map h (reverse l) = reverse (map h l)
```

Induktionsschritt:

2. `map h (reverse (a : as)) = reverse (map h (a : as)) ?`

```
map h (reverse (a : as))
= map h ((reverse as) ++ [a])
= (map h (reverse as)) ++ (map h [a])
= (reverse (map h as)) ++ [h a]
```

Manchmal nicht so einfach

Naives Umdrehen einer Liste:

```
reverse :: [a] → [a]
reverse []      = []
reverse (a : as) = (reverse as) ++ [a]
```

Zu beweisen:

```
map h (reverse l) = reverse (map h l)
```

Induktionsschritt:

2. `map h (reverse (a : as)) = reverse (map h (a : as)) ?`

```
map h (reverse (a : as))
= map h ((reverse as) ++ [a])
= (map h (reverse as)) ++ (map h [a])
= (reverse (map h as)) ++ [h a]
= reverse ((h a) : (map h as))
```

Manchmal nicht so einfach

Naives Umdrehen einer Liste:

```
reverse :: [a] → [a]
reverse []      = []
reverse (a : as) = (reverse as) ++ [a]
```

Zu beweisen:

$$\text{map } h (\text{reverse } l) = \text{reverse } (\text{map } h l)$$

Induktionsschritt:

2. $\text{map } h (\text{reverse } (a : as)) = \text{reverse } (\text{map } h (a : as))$?

```
map h (reverse (a : as))
= map h ((reverse as) ++ [a])
= (map h (reverse as)) ++ (map h [a])
= (reverse (map h as)) ++ [h a]
= reverse ((h a) : (map h as))
= reverse (map h (a : as))
```

Manchmal nicht so einfach

Naives Umdrehen einer Liste:

```
reverse :: [a] → [a]
reverse []      = []
reverse (a : as) = (reverse as) ++ [a]
```

Zu beweisen:

$$\text{map } h (\text{reverse } l) = \text{reverse } (\text{map } h l)$$

Induktionsschritt:

2. $\text{map } h (\text{reverse } (a : as)) = \text{reverse } (\text{map } h (a : as))$?
OK

Manchmal nicht so einfach

Effizientes Umdrehen einer Liste:

`reverse'` :: [a] → [a] → [a]

`reverse'` [] *bs* = *bs*

`reverse'` (a : as) *bs* = `reverse'` as (a : *bs*)

Manchmal nicht so einfach

Effizientes Umdrehen einer Liste:

`reverse'` :: $[a] \rightarrow [a] \rightarrow [a]$

`reverse'` [] $bs = bs$

`reverse'` (a : as) $bs = \text{reverse}' as (a : bs)$

Zu beweisen:

$\text{map } h (\text{reverse}' l []) = \text{reverse}' (\text{map } h l) []$

Manchmal nicht so einfach

Effizientes Umdrehen einer Liste:

`reverse'` :: $[a] \rightarrow [a] \rightarrow [a]$

`reverse'` [] $bs = bs$

`reverse'` (a : as) $bs = \text{reverse}' as (a : bs)$

Zu beweisen:

$\text{map } h (\text{reverse}' l []) = \text{reverse}' (\text{map } h l) []$

Induktionsschritt:

2. $\text{map } h (\text{reverse}' (a : as) []) = \text{reverse}' (\text{map } h (a : as)) [] ?$

Manchmal nicht so einfach

Effizientes Umdrehen einer Liste:

`reverse'` :: $[a] \rightarrow [a] \rightarrow [a]$

`reverse'` [] $bs = bs$

`reverse'` (a : as) $bs = \text{reverse}' as (a : bs)$

Zu beweisen:

$\text{map } h (\text{reverse}' l []) = \text{reverse}' (\text{map } h l) []$

Induktionsschritt:

2. $\text{map } h (\text{reverse}' (a : as) []) = \text{reverse}' (\text{map } h (a : as)) [] ?$
 $\text{map } h (\text{reverse}' (a : as) [])$

Manchmal nicht so einfach

Effizientes Umdrehen einer Liste:

`reverse'` :: $[a] \rightarrow [a] \rightarrow [a]$

`reverse'` [] $bs = bs$

`reverse'` (a : as) $bs = \text{reverse}' as (a : bs)$

Zu beweisen:

$$\text{map } h (\text{reverse}' l []) = \text{reverse}' (\text{map } h l) []$$

Induktionsschritt:

2. $\text{map } h (\text{reverse}' (a : as) []) = \text{reverse}' (\text{map } h (a : as)) [] ?$

$$\begin{aligned} & \text{map } h (\text{reverse}' (a : as) []) \\ &= \text{map } h (\text{reverse}' as [a]) \end{aligned}$$

Manchmal nicht so einfach

Effizientes Umdrehen einer Liste:

`reverse'` :: `[a] → [a] → [a]`

`reverse'` [] `bs` = `bs`

`reverse'` (`a : as`) `bs` = `reverse'` `as` (`a : bs`)

Zu beweisen:

$$\text{map } h (\text{reverse}' l []) = \text{reverse}' (\text{map } h l) []$$

Induktionsschritt:

2. $\text{map } h (\text{reverse}' (a : as) []) = \text{reverse}' (\text{map } h (a : as)) [] ?$

$$= \text{map } h (\text{reverse}' (a : as) [])$$

$$= \text{map } h (\text{reverse}' as [a])$$

?

Manchmal nicht so einfach

Effizientes Umdrehen einer Liste:

`reverse'` :: $[a] \rightarrow [a] \rightarrow [a]$

`reverse'` [] $bs = bs$

`reverse'` (a : as) $bs = \text{reverse}' as (a : bs)$

Zu beweisen:

$$\text{map } h (\text{reverse}' l []) = \text{reverse}' (\text{map } h l) []$$

Induktionsschritt:

2. $\text{map } h (\text{reverse}' (a : as) []) = \text{reverse}' (\text{map } h (a : as)) [] ?$

$$= \text{map } h (\text{reverse}' (a : as) [])$$

$$= \text{map } h (\text{reverse}' as [a])$$

?

$$\text{reverse}' (\text{map } h (a : as)) []$$

Manchmal nicht so einfach

Effizientes Umdrehen einer Liste:

$\text{reverse}' :: [a] \rightarrow [a] \rightarrow [a]$

$\text{reverse}' [] \quad bs = bs$

$\text{reverse}' (a : as) \quad bs = \text{reverse}' as (a : bs)$

Zu beweisen:

$$\text{map } h (\text{reverse}' l []) = \text{reverse}' (\text{map } h l) []$$

Induktionsschritt:

2. $\text{map } h (\text{reverse}' (a : as) []) = \text{reverse}' (\text{map } h (a : as)) [] ?$

$$= \text{map } h (\text{reverse}' (a : as) [])$$

$$= \text{map } h (\text{reverse}' as [a])$$

?

$$= \text{reverse}' ((h a) : (\text{map } h as)) []$$

$$= \text{reverse}' (\text{map } h (a : as)) []$$

Manchmal nicht so einfach

Effizientes Umdrehen einer Liste:

`reverse'` :: $[a] \rightarrow [a] \rightarrow [a]$

`reverse'` [] $bs = bs$

`reverse'` (a : as) $bs = \text{reverse}' as (a : bs)$

Zu beweisen:

$$\text{map } h (\text{reverse}' l []) = \text{reverse}' (\text{map } h l) []$$

Induktionsschritt:

2. $\text{map } h (\text{reverse}' (a : as) []) = \text{reverse}' (\text{map } h (a : as)) [] ?$

$$= \text{map } h (\text{reverse}' (a : as) [])$$

$$= \text{map } h (\text{reverse}' as [a])$$

?

$$\text{reverse}' (\text{map } h as) [h a]$$

$$= \text{reverse}' ((h a) : (\text{map } h as)) []$$

$$= \text{reverse}' (\text{map } h (a : as)) []$$

Manchmal nicht so einfach

Effizientes Umdrehen einer Liste:

`reverse'` :: $[a] \rightarrow [a] \rightarrow [a]$

`reverse'` [] $bs = bs$

`reverse'` (a : as) $bs = \text{reverse}' as (a : bs)$

Zu beweisen:

$$\text{map } h (\text{reverse}' l []) = \text{reverse}' (\text{map } h l) []$$

Induktionsschritt:

2. $\text{map } h (\text{reverse}' (a : as) []) = \text{reverse}' (\text{map } h (a : as)) []$?

$$= \text{map } h (\text{reverse}' (a : as) [])$$

$$= \text{map } h (\text{reverse}' as [a])$$

?

$$= \text{reverse}' (\text{map } h as) [h a]$$

$$= \text{reverse}' ((h a) : (\text{map } h as)) []$$

$$= \text{reverse}' (\text{map } h (a : as)) []$$

Freie Theoreme [Wadler, 1989]

Ursprüngliches Beispiel:

```
filter :: (a -> Bool) -> [a] -> [a]
filter p []           = []
filter p (a : as)    | p a           = a : (filter p as)
                    | otherwise     = filter p as
```

Für jede Wahl von p , h und l gilt:

```
filter p (map h l) = map h (filter (p ∘ h) l)
```

Bewiesen per Induktion.

Freie Theoreme [Wadler, 1989]

Ursprüngliches Beispiel:

```
filter :: (a → Bool) → [a] → [a]
filter p []           = []
filter p (a : as) | p a           = a : (filter p as)
                  | otherwise = filter p as
```

Für jede Wahl von p , h und l gilt:

```
filter p (map h l) = map h (filter (p ∘ h) l)
```

Bewiesen per Induktion.

Aber auch möglich **ohne** Induktion!

Freie Theoreme [Wadler, 1989]

Ursprüngliches Beispiel:

`filter` :: $(a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]$

Für jede Wahl von p , h und l gilt:

`filter` p (`map` h l) = `map` h (`filter` ($p \circ h$) l)

Bewiesen per Induktion.

Aber auch möglich **ohne** Induktion!

Freie Theoreme [Wadler, 1989]

Ursprüngliches Beispiel:

`filter` :: $(a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]$

`takeWhile` :: $(a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]$

Für jede Wahl von p , h und l gilt:

`filter` p (`map` h l) = `map` h (`filter` ($p \circ h$) l)

`takeWhile` p (`map` h l) = `map` h (`takeWhile` ($p \circ h$) l)

Freie Theoreme [Wadler, 1989]

Ursprüngliches Beispiel:

`filter` :: $(a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]$

`takeWhile` :: $(a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]$

`g` :: $(a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]$

Für jede Wahl von p , h und l gilt:

`filter` p (`map` h l) = `map` h (`filter` ($p \circ h$) l)

`takeWhile` p (`map` h l) = `map` h (`takeWhile` ($p \circ h$) l)

`g` p (`map` h l) = `map` h (`g` ($p \circ h$) l)

Freie Theoreme [Wadler, 1989]

Ursprüngliches Beispiel:

`filter` :: $(a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]$

`takeWhile` :: $(a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]$

`g` :: $(a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]$

Für jede Wahl von p , h und l gilt:

`filter` p (`map` h l) = `map` h (`filter` ($p \circ h$) l)

`takeWhile` p (`map` h l) = `map` h (`takeWhile` ($p \circ h$) l)

`g` p (`map` h l) = `map` h (`g` ($p \circ h$) l)

Aber wie soll man so etwas beweisen können?

Versuch einer Argumentation

- $g :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]$ muss für jede mögliche Instanziierung von a einheitlich arbeiten.

Versuch einer Argumentation

- $g :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]$ muss für jede mögliche Instanziierung von a einheitlich arbeiten.
- Die Ausgabeliste kann nur **Elemente der Eingabe /** enthalten.

Versuch einer Argumentation

- $g :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]$ muss für jede mögliche Instanziierung von a einheitlich arbeiten.
- Die Ausgabeliste kann nur **Elemente der Eingabe I** enthalten.
- Welche, und in welcher Reihenfolge/Vielfachheit, kann lediglich **von I und dem Eingabepredikat p abhängen.**

Versuch einer Argumentation

- $g :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]$ muss für jede mögliche Instanziierung von a einheitlich arbeiten.
- Die Ausgabeliste kann nur Elemente der Eingabe l enthalten.
- Welche, und in welcher Reihenfolge/Vielfachheit, kann lediglich von l und dem Eingabepredikat p abhängen.
- Die einzig möglichen Grundlagen zur Entscheidung sind die Länge von l und die Ergebnisse von p auf Elementen von l .

Versuch einer Argumentation

- $g :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]$ muss für jede mögliche Instanziierung von a einheitlich arbeiten.
- Die Ausgabeliste kann nur Elemente der Eingabe l enthalten.
- Welche, und in welcher Reihenfolge/Vielfachheit, kann lediglich von l und dem Eingabepredikat p abhängen.
- Die einzig möglichen Grundlagen zur Entscheidung sind die **Länge von l** und die Ergebnisse von p auf Elementen von l .
- Aber, die Listen $(\text{map } h \ l)$ und l haben stets **die selbe Länge**.

Versuch einer Argumentation

- $g :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]$ muss für jede mögliche Instanziierung von a einheitlich arbeiten.
- Die Ausgabeliste kann nur Elemente der Eingabe l enthalten.
- Welche, und in welcher Reihenfolge/Vielfachheit, kann lediglich von l und dem Eingabepredikat p abhängen.
- Die einzig möglichen Grundlagen zur Entscheidung sind die Länge von l und die **Ergebnisse von p auf Elementen von l** .
- Aber, die Listen $(\text{map } h \ l)$ und l haben stets die selbe Länge.
- Und Anwendung von p auf ein Element von $(\text{map } h \ l)$ hat stets **das selbe Ergebnis** wie Anwendung von $(p \circ h)$ auf das entsprechende Element von l .

Versuch einer Argumentation

- $g :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]$ muss für jede mögliche Instanziierung von a einheitlich arbeiten.
- Die Ausgabeliste kann nur Elemente der Eingabe l enthalten.
- Welche, und in welcher Reihenfolge/Vielfachheit, kann lediglich von l und dem Eingabepredikat p abhängen.
- Die einzig möglichen Grundlagen zur Entscheidung sind die Länge von l und die Ergebnisse von p auf Elementen von l .
- Aber, die Listen $(\text{map } h \ l)$ und l haben stets die selbe Länge.
- Und Anwendung von p auf ein Element von $(\text{map } h \ l)$ hat stets das selbe Ergebnis wie Anwendung von $(p \circ h)$ auf das entsprechende Element von l .
- Also wählt g mit p stets „die selben“ Elemente aus $(\text{map } h \ l)$ wie es g mit $(p \circ h)$ aus l tut,

Versuch einer Argumentation

- $g :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]$ muss für jede mögliche Instanziierung von a einheitlich arbeiten.
- Die Ausgabeliste kann nur Elemente der Eingabe l enthalten.
- Welche, und in welcher Reihenfolge/Vielfachheit, kann lediglich von l und dem Eingabepredikat p abhängen.
- Die einzig möglichen Grundlagen zur Entscheidung sind die Länge von l und die Ergebnisse von p auf Elementen von l .
- Aber, die Listen $(\text{map } h \ l)$ und l haben stets die selbe Länge.
- Und Anwendung von p auf ein Element von $(\text{map } h \ l)$ hat stets das selbe Ergebnis wie Anwendung von $(p \circ h)$ auf das entsprechende Element von l .
- Also wählt g mit p stets „die selben“ Elemente aus $(\text{map } h \ l)$ wie es g mit $(p \circ h)$ aus l tut, **außer dass im ersten Fall die entsprechenden Abbilder unter h ausgegeben werden.**

Versuch einer Argumentation

- $g :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]$ muss für jede mögliche Instanziierung von a einheitlich arbeiten.
- Die Ausgabeliste kann nur Elemente der Eingabe l enthalten.
- Welche, und in welcher Reihenfolge/Vielfachheit, kann lediglich von l und dem Eingabepredikat p abhängen.
- Die einzig möglichen Grundlagen zur Entscheidung sind die Länge von l und die Ergebnisse von p auf Elementen von l .
- Aber, die Listen $(\text{map } h \ l)$ und l haben stets die selbe Länge.
- Und Anwendung von p auf ein Element von $(\text{map } h \ l)$ hat stets das selbe Ergebnis wie Anwendung von $(p \circ h)$ auf das entsprechende Element von l .
- Also wählt g mit p stets „die selben“ Elemente aus $(\text{map } h \ l)$ wie es g mit $(p \circ h)$ aus l tut, außer dass im ersten Fall die entsprechenden **Abbilder unter h** ausgegeben werden.
- Also ist $(g \ p \ (\text{map } h \ l))$ gleich $(\text{map } h \ (g \ (p \circ h) \ l))$.

Versuch einer Argumentation

- $g :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]$ muss für jede mögliche Instanziierung von a einheitlich arbeiten.
- Die Ausgabeliste kann nur Elemente der Eingabe l enthalten.
- Welche, und in welcher Reihenfolge/Vielfachheit, kann lediglich von l und dem Eingabepredikat p abhängen.
- Die einzig möglichen Grundlagen zur Entscheidung sind die Länge von l und die Ergebnisse von p auf Elementen von l .
- Aber, die Listen $(\text{map } h \ l)$ und l haben stets die selbe Länge.
- Und Anwendung von p auf ein Element von $(\text{map } h \ l)$ hat stets das selbe Ergebnis wie Anwendung von $(p \circ h)$ auf das entsprechende Element von l .
- Also wählt g mit p stets „die selben“ Elemente aus $(\text{map } h \ l)$ wie es g mit $(p \circ h)$ aus l tut, außer dass im ersten Fall die entsprechenden Abbilder unter h ausgegeben werden.
- Also ist $(g \ p \ (\text{map } h \ l))$ gleich $(\text{map } h \ (g \ (p \circ h) \ l))$.
- **Genau das wollten wir beweisen!**

Geht es vielleicht etwas formaler?

Gegeben:

$$g :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]$$

Behauptung, für jede Wahl von p , h und l :

$$g\ p\ (\text{map}\ h\ l) = \text{map}\ h\ (g\ (p \circ h)\ l)$$

Geht es vielleicht etwas formaler?

Gegeben:

$$g :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]$$

Behauptung, für jede Wahl von p , h und l :

$$g \ p \ (\text{map } h \ l) = \text{map } h \ (g \ (p \circ h) \ l)$$

Wie wäre es mit Induktion über den Syntaxaufbau von g ?

Geht es vielleicht etwas formaler?

Gegeben:

$$g :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]$$

Behauptung, für jede Wahl von p , h und l :

$$g\ p\ (\text{map}\ h\ l) = \text{map}\ h\ (g\ (p \circ h)\ l)$$

Wie wäre es mit Induktion über den Syntaxaufbau von g ?

Problem: lässt Typinformation außer Acht

Geht es vielleicht etwas formaler?

Gegeben:

$$g :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]$$

Behauptung, für jede Wahl von p , h und l :

$$g\ p\ (\text{map}\ h\ l) = \text{map}\ h\ (g\ (p \circ h)\ l)$$

Wie wäre es mit Induktion über den Syntaxaufbau von g ?

Problem: lässt Typinformation außer Acht

Wie wäre es dann mit Induktion nur über alle Funktionen des Typs $(a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]$?

Geht es vielleicht etwas formaler?

Gegeben:

$$g :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]$$

Behauptung, für jede Wahl von p , h und l :

$$g \ p \ (\text{map } h \ l) = \text{map } h \ (g \ (p \circ h) \ l)$$

Wie wäre es mit Induktion über den Syntaxaufbau von g ?

Problem: lässt Typinformation außer Acht

Wie wäre es dann mit Induktion nur über alle Funktionen des Typs $(a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]$?

Problem: dieser Bereich nicht kompositionell

Geht es vielleicht etwas formaler?

Gegeben:

$$g :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]$$

Behauptung, für jede Wahl von p , h und l :

$$g\ p\ (\text{map}\ h\ l) = \text{map}\ h\ (g\ (p \circ h)\ l)$$

Wie wäre es mit Induktion über den Syntaxaufbau von g ?

Problem: lässt Typinformation außer Acht

Wie wäre es dann mit Induktion nur über alle Funktionen des Typs $(a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]$?

Problem: dieser Bereich nicht kompositionell

Wir müssen also allgemein Aussagen der obigen Art für **alle** Typen „gleichzeitig“ beweisen!

Geht es vielleicht etwas formaler?

Strategie:

- zu jedem Typ angeben, welche möglichen Varianten (semantisch, nicht syntaktisch) von Elementen diesen Typs es gibt

Geht es vielleicht etwas formaler?

Strategie:

- zu jedem Typ angeben, welche möglichen Varianten (semantisch, nicht syntaktisch) von Elementen diesen Typs es gibt
- das Ganze möglichst kompositionell über den Aufbau von Typen

Geht es vielleicht etwas formaler?

Strategie:

- zu jedem Typ angeben, welche möglichen Varianten (semantisch, nicht syntaktisch) von Elementen diesen Typs es gibt
- das Ganze möglichst kompositionell über den Aufbau von Typen
- später per Induktion über die Syntax beweisen, dass jede Funktion die semantischen Einschränkungen ihres Typs erfüllt

Geht es vielleicht etwas formaler?

Strategie:

- zu jedem Typ angeben, welche möglichen Varianten (semantisch, nicht syntaktisch) von Elementen diesen Typs es gibt
- das Ganze möglichst kompositionell über den Aufbau von Typen
- später per Induktion über die Syntax beweisen, dass jede Funktion die semantischen Einschränkungen ihres Typs erfüllt

Erster Punkt an Beispielen

Frage: Welche Funktionen haben den Typ „ $(a, a) \rightarrow a$ “ ?

Erster Punkt an Beispielen

Frage: Welche Funktionen haben den Typ „ $(a, a) \rightarrow a$ “ ?

Antwort: nur `fst` und `snd`

Erster Punkt an Beispielen

Frage: Welche Funktionen haben den Typ „ $(a, a) \rightarrow a$ “ ?

Antwort: nur `fst` und `snd` (und semantisch äquivalente)

Erster Punkt an Beispielen

Frage: Welche Funktionen haben den Typ „ $(a, a) \rightarrow a$ “ ?

Antwort: nur `fst` und `snd` (und semantisch äquivalente)

Frage: Welche Funktionen haben den Typ „ $a \rightarrow a$ “ ?

Erster Punkt an Beispielen

Frage: Welche Funktionen haben den Typ „ $(a, a) \rightarrow a$ “ ?

Antwort: nur `fst` und `snd` (und semantisch äquivalente)

Frage: Welche Funktionen haben den Typ „ $a \rightarrow a$ “ ?

Antwort: nur `id`

Erster Punkt an Beispielen

Frage: Welche Funktionen haben den Typ „ $(a, a) \rightarrow a$ “ ?

Antwort: nur `fst` und `snd` (und semantisch äquivalente)

Frage: Welche Funktionen haben den Typ „ $a \rightarrow a$ “ ?

Antwort: nur `id`

Frage: Welche Funktionen haben den Typ „ $a \rightarrow [a]$ “ ?

Erster Punkt an Beispielen

Frage: Welche Funktionen haben den Typ „ $(a, a) \rightarrow a$ “ ?

Antwort: nur `fst` und `snd` (und semantisch äquivalente)

Frage: Welche Funktionen haben den Typ „ $a \rightarrow a$ “ ?

Antwort: nur `id`

Frage: Welche Funktionen haben den Typ „ $a \rightarrow [a]$ “ ?

Antwort: `f0`, `f1`, `f2`, ..., wobei:

`f0` = []

Erster Punkt an Beispielen

Frage: Welche Funktionen haben den Typ „ $(a, a) \rightarrow a$ “ ?

Antwort: nur `fst` und `snd` (und semantisch äquivalente)

Frage: Welche Funktionen haben den Typ „ $a \rightarrow a$ “ ?

Antwort: nur `id`

Frage: Welche Funktionen haben den Typ „ $a \rightarrow [a]$ “ ?

Antwort: `f0`, `f1`, `f2`, ..., wobei:

$$f_0 _ = []$$

$$f_1 \ a = [a]$$

Erster Punkt an Beispielen

Frage: Welche Funktionen haben den Typ „ $(a, a) \rightarrow a$ “ ?

Antwort: nur `fst` und `snd` (und semantisch äquivalente)

Frage: Welche Funktionen haben den Typ „ $a \rightarrow a$ “ ?

Antwort: nur `id`

Frage: Welche Funktionen haben den Typ „ $a \rightarrow [a]$ “ ?

Antwort: `f0`, `f1`, `f2`, ..., wobei:

`f0` `_` = `[]`

`f1` `a` = `[a]`

`f2` `a` = `[a, a]`

...

Erster Punkt an Beispielen

Frage: Welche Funktionen haben den Typ „ $(a, a) \rightarrow a$ “ ?

Antwort: nur `fst` und `snd` (und semantisch äquivalente)

Frage: Welche Funktionen haben den Typ „ $a \rightarrow a$ “ ?

Antwort: nur `id`

Frage: Welche Funktionen haben den Typ „ $a \rightarrow [a]$ “ ?

Antwort: `f0`, `f1`, `f2`, ..., wobei:

`f0` `_` = []

`f1` `a` = [a]

`f2` `a` = [a, a]

...

Frage: Welche Funktionen haben den Typ „ $[a] \rightarrow [a]$ “ ?

Welche Möglichkeiten gibt es für $g :: [a] \rightarrow [a]$?

Zunächst wieder intuitiv:

Welche Möglichkeiten gibt es für $g :: [a] \rightarrow [a]$?

Zunächst wieder intuitiv:

- Die Ausgabe- kann nur **Elemente der Eingabeliste** enthalten.

Welche Möglichkeiten gibt es für $g :: [a] \rightarrow [a]$?

Zunächst wieder intuitiv:

- Die Ausgabe- kann nur **Elemente der Eingabeliste** enthalten.
- Welche, und in welcher Reihenfolge/Vielfachheit, kann lediglich **von dieser Liste abhängen**

Welche Möglichkeiten gibt es für $g :: [a] \rightarrow [a]$?

Zunächst wieder intuitiv:

- Die Ausgabe- kann nur Elemente der Eingabeliste enthalten.
- Welche, und in welcher Reihenfolge/Vielfachheit, kann lediglich von dieser Liste abhängen, und zwar von ihrer Länge.

Welche Möglichkeiten gibt es für $g :: [a] \rightarrow [a]$?

Zunächst wieder intuitiv:

- Die Ausgabe- kann nur Elemente der Eingabeliste enthalten.
- Welche, und in welcher Reihenfolge/Vielfachheit, kann lediglich **von dieser Liste abhängen, und zwar von ihrer Länge.**
- Folglich sind für eine feste Eingabelänge die Länge der Ausgabeliste sowie die Anordnung der Elemente darin stets fix.

Welche Möglichkeiten gibt es für $g :: [a] \rightarrow [a]$?

Zunächst wieder intuitiv:

- Die Ausgabe- kann nur Elemente der Eingabeliste enthalten.
- Welche, und in welcher Reihenfolge/Vielfachheit, kann lediglich von dieser Liste abhängen, und zwar von ihrer Länge.
- Folglich sind für eine feste Eingabelänge die Länge der Ausgabeliste sowie die Anordnung der Elemente darin stets fix.

Formal beschrieben:

Für jedes $g :: [a] \rightarrow [a]$ gibt es:

Welche Möglichkeiten gibt es für $g :: [a] \rightarrow [a]$?

Zunächst wieder intuitiv:

- Die Ausgabe- kann nur Elemente der Eingabeliste enthalten.
- Welche, und in welcher Reihenfolge/Vielfachheit, kann lediglich von dieser Liste abhängen, und zwar von ihrer Länge.
- Folglich sind für eine feste Eingabelänge die **Länge der Ausgabeliste** sowie die Anordnung der Elemente darin stets fix.

Formal beschrieben:

Für jedes $g :: [a] \rightarrow [a]$ gibt es:

- **eine Funktion $f : \mathbb{N} \rightarrow \mathbb{N}$, so dass für jedes $n \in \mathbb{N}$ aus einer Liste der Länge n mit g stets eine Liste der Länge $f(n)$ wird;**

Welche Möglichkeiten gibt es für $g :: [a] \rightarrow [a]$?

Zunächst wieder intuitiv:

- Die Ausgabe- kann nur Elemente der Eingabeliste enthalten.
- Welche, und in welcher Reihenfolge/Vielfachheit, kann lediglich von dieser Liste abhängen, und zwar von ihrer Länge.
- Folglich sind für eine feste Eingabelänge die Länge der Ausgabeliste sowie die **Anordnung der Elemente darin** stets fix.

Formal beschrieben:

Für jedes $g :: [a] \rightarrow [a]$ gibt es:

- eine Funktion $f : \mathbb{N} \rightarrow \mathbb{N}$, so dass für jedes $n \in \mathbb{N}$ aus einer Liste der Länge n mit g stets eine Liste der Länge $f(n)$ wird;
- eine Funktion $h : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$, so dass für jede Liste einer Länge $n \in \mathbb{N}$ und für jedes $0 \leq m < f(n)$, das m -te Element der Ausgabeliste das $h(n, m)$ -te Element der Eingabeliste ist.

Welche Möglichkeiten gibt es für $g :: [a] \rightarrow [a]$?

Zunächst wieder intuitiv:

- Die Ausgabe- kann nur Elemente der Eingabeliste enthalten.
- Welche, und in welcher Reihenfolge/Vielfachheit, kann lediglich von dieser Liste abhängen, und zwar von ihrer Länge.
- Folglich sind für eine feste Eingabelänge die Länge der Ausgabeliste sowie die Anordnung der Elemente darin stets fix.

Formal beschrieben:

Für jedes $g :: [a] \rightarrow [a]$ gibt es:

- eine Funktion $f : \mathbb{N} \rightarrow \mathbb{N}$, so dass für jedes $n \in \mathbb{N}$ aus einer Liste der Länge n mit g stets eine Liste der Länge $f(n)$ wird;
- eine Funktion $h : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$, so dass für jede Liste einer Länge $n \in \mathbb{N}$ und für jedes $0 \leq m < f(n)$, das m -te Element der Ausgabeliste das $h(n, m)$ -te Element der Eingabeliste ist.

Geht es vielleicht etwas formaler?

Strategie:

- zu jedem Typ angeben, welche möglichen Varianten (semantisch, nicht syntaktisch) von Elementen diesen Typs es gibt
- das Ganze möglichst kompositionell über den Aufbau von Typen
- später per Induktion über die Syntax beweisen, dass jede Funktion die semantischen Einschränkungen ihres Typs erfüllt