

# Die Komplement-Funktion

```
type IntMap  $\alpha$  = [(Int,  $\alpha$ )]
```

```
compl :: [ $\alpha$ ]  $\rightarrow$  (Int, IntMap  $\alpha$ )
```

```
compl s = let n = (length s) - 1
```

```
    t = [0..n]
```

```
    g = zip t s
```

```
    g' = filter ( $\lambda(i, \_)$   $\rightarrow$  notElem i (get t)) g
```

```
  in (n + 1, g')
```

Zum Beispiel:

```
get = tail       $\rightsquigarrow$  compl "abcde" = (5, [(0, 'a')])
```

```
get = take 3     $\rightsquigarrow$  compl "abcde" = (5, [(3, 'd'), (4, 'e')])
```

```
get = reverse   $\rightsquigarrow$  compl "abcde" = (5, [])
```

## Ein Inverses zu $\lambda s \rightarrow (\text{get } s, \text{compl } s)$

`inv` ::  $([\alpha], (\text{Int}, \text{IntMap } \alpha)) \rightarrow [\alpha]$

```
inv (v', (n + 1, g')) = let t = [0..n]
                          h = assoc† (get t) v'
                          h' = h ++ g'
                      in seq h (map (\i → fromJust (lookup i h')) t)
```

Zum Beispiel:

`get = tail`     $\rightsquigarrow$     `inv ("bcde", (5, [(0, 'a')])) = "abcde"`

`get = take 3`     $\rightsquigarrow$     `inv ("xyz", (5, [(3, 'd'), (4, 'e')])) = "xyzde"`

Formal zu beweisen:

- `inv (get s, compl s) = s`
- wenn `inv (v, c)` definiert, dann `get (inv (v, c)) = v`
- wenn `inv (v, c)` definiert, dann `compl (inv (v, c)) = c`

<sup>†</sup> Für den Moment, kann als `zip` angenommen werden.

## Insgesamt:

```
type IntMap  $\alpha$  = [(Int,  $\alpha$ )]
```

```
compl :: [ $\alpha$ ]  $\rightarrow$  (Int, IntMap  $\alpha$ )
```

```
compl s = let n = (length s) - 1  
            t = [0..n]  
            g = zip t s  
            g' = filter ( $\lambda(i, \_) \rightarrow$  notElem i (get t)) g  
            in (n + 1, g')
```

```
inv :: ([ $\alpha$ ], (Int, IntMap  $\alpha$ ))  $\rightarrow$  [ $\alpha$ ]
```

```
inv (v', (n + 1, g')) = let t = [0..n]  
                        h = assoc (get t) v'  
                        h' = h ++ g'  
                        in seq h (map ( $\lambda i \rightarrow$  fromJust (lookup i h')) t)
```

```
put :: [ $\alpha$ ]  $\rightarrow$  [ $\alpha$ ]  $\rightarrow$  [ $\alpha$ ]
```

```
put s v' = inv (v', compl s)
```

## „Fusion“

Inlining von `compl` und `inv` in `put`:

```
put s v' = let n = (length s) - 1
            t = [0..n]
            g = zip t s
            g' = filter (\(i, _) → notElem i (get t)) g
            h = assoc (get t) v'
            h' = h ++ g'
        in seq h (map (\i → fromJust (lookup i h')) t)
```

## „Fusion“

Inlining von `compl` und `inv` in `put`:

```
put s v' = let n = (length s) - 1
            t = [0..n]
            g = zip t s
            g' = filter (\(i, _) → notElem i (get t)) g
            h = assoc (get t) v'
            h' = h ++ g'
        in seq h (map (\i → fromJust (lookup i h')) t)
```

```
assoc [] [] = []
assoc (i : is) (b : bs) = let m = assoc is bs
                          in case lookup i m of
                              Nothing → (i, b) : m
                              Just c | b == c → m
```

## „Fusion“

Inlining von `compl` und `inv` in `put`:

```
bff get s v' = let n = (length s) - 1
                t = [0..n]
                g = zip t s
                g' = filter (\(i, _) → notElem i (get t)) g
                h = assoc (get t) v'
                h' = h ++ g'
            in seq h (map (\i → fromJust (lookup i h')) t)
```

```
assoc [] [] = []
assoc (i : is) (b : bs) = let m = assoc is bs
                            in case lookup i m of
                                Nothing → (i, b) : m
                                Just c | b == c → m
```

## „Fusion“

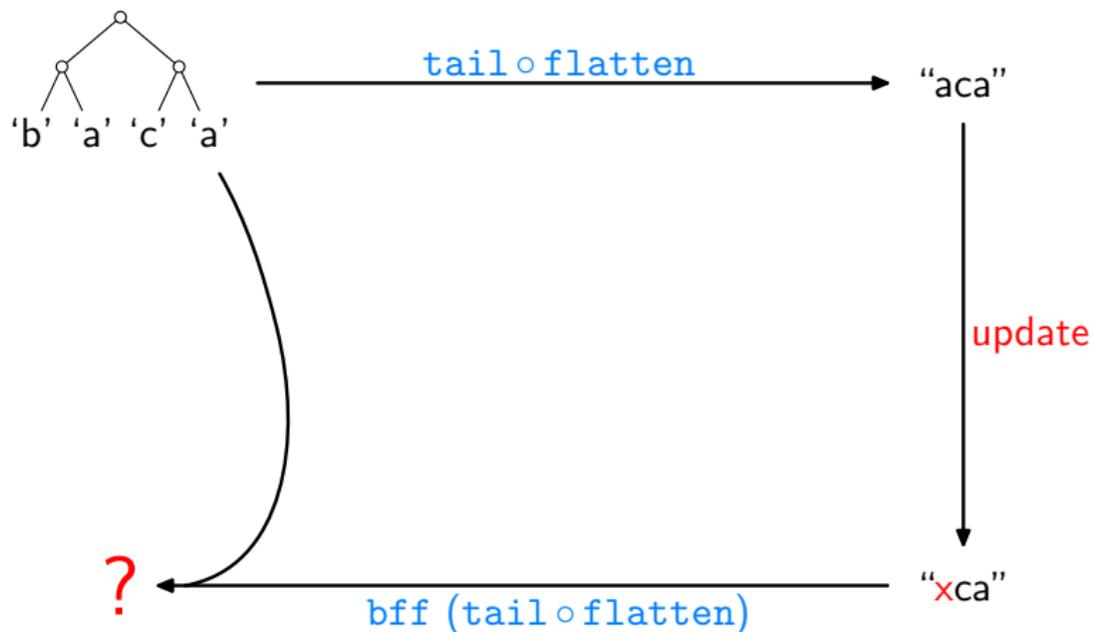
Inlining von `compl` und `inv` in `put`:

```
bff get s v' = let n = (length s) - 1
                 t = [0..n]
                 g = zip t s
                 g' = filter (\(i, _) → notElem i (get t)) g
                 h = assoc (get t) v'
                 h' = h ++ g'
             in seq h (map (\i → fromJust (lookup i h')) t)
```

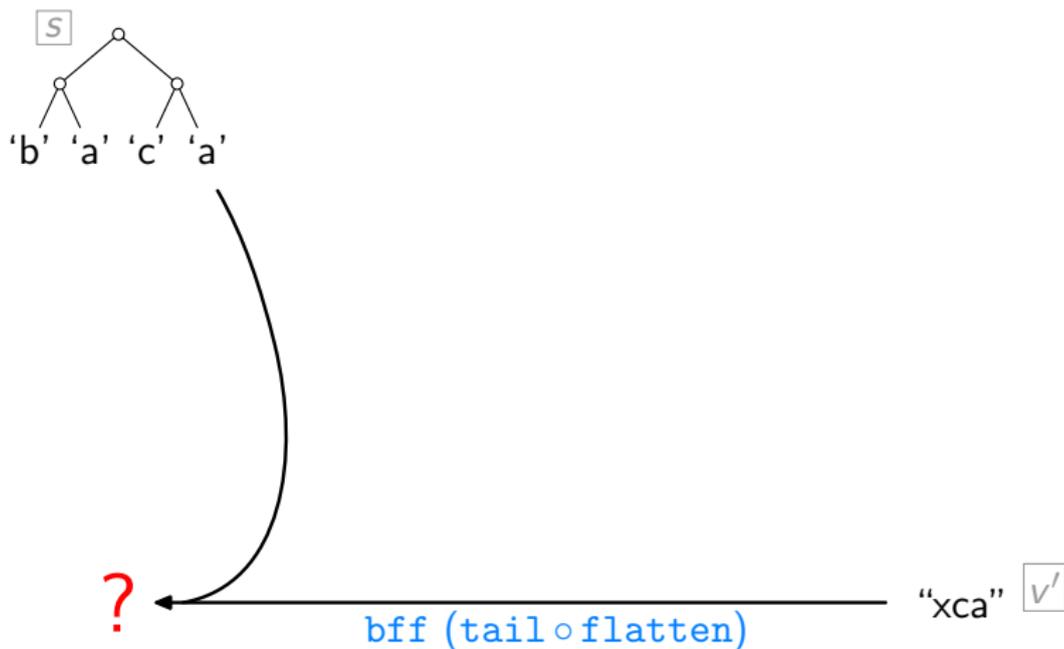
```
assoc [] [] = []
assoc (i : is) (b : bs) = let m = assoc is bs
                           in case lookup i m of
                               Nothing → (i, b) : m
                               Just c | b == c → m
```

Tatsächlicher Code nur geringfügig ausgeklügelter!

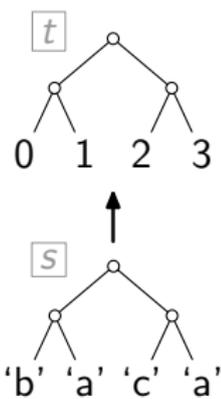
# Die resultierende Bidirektionalisierungsmethode in Aktion



# Die resultierende Bidirektionalisierungsmethode in Aktion

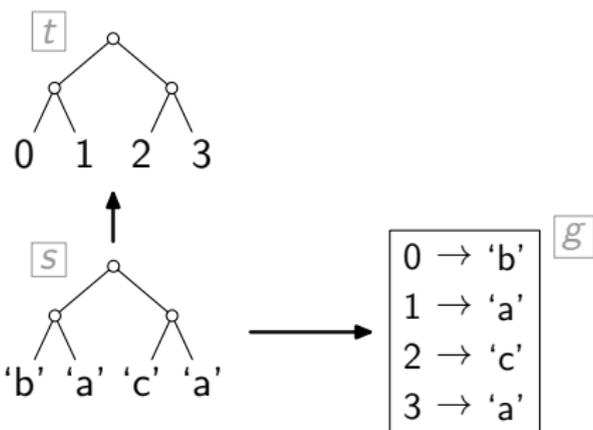


# Die resultierende Bidirektionalisierungsmethode in Aktion



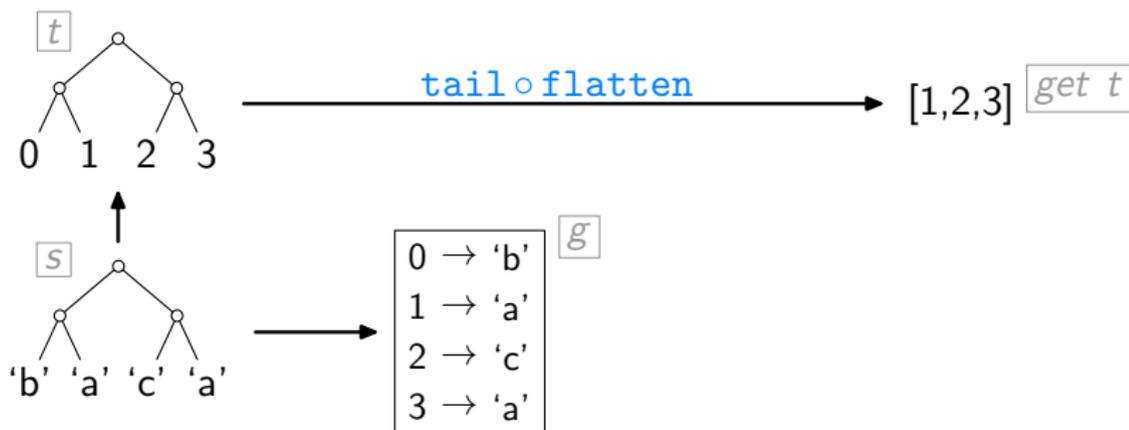
“xca”  $v'$

# Die resultierende Bidirektionalisierungsmethode in Aktion



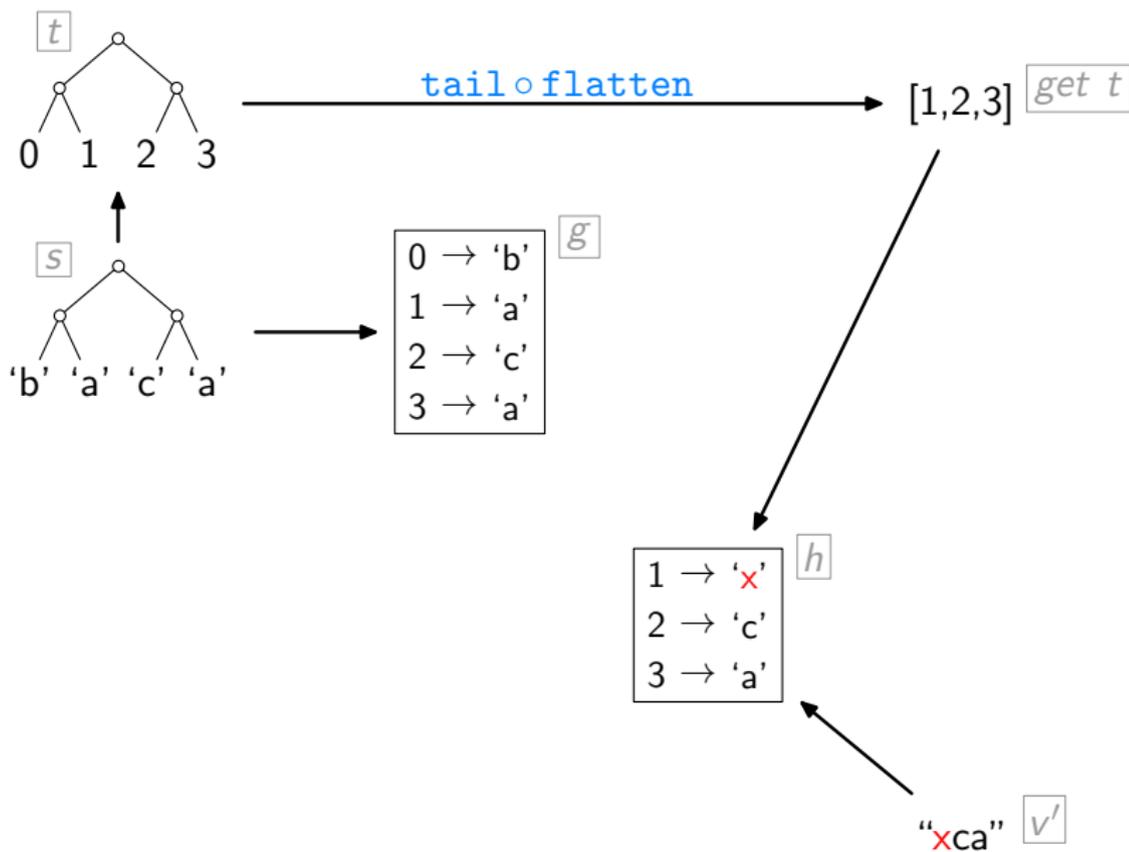
“xca”  $v'$

# Die resultierende Bidirektionalisierungsmethode in Aktion

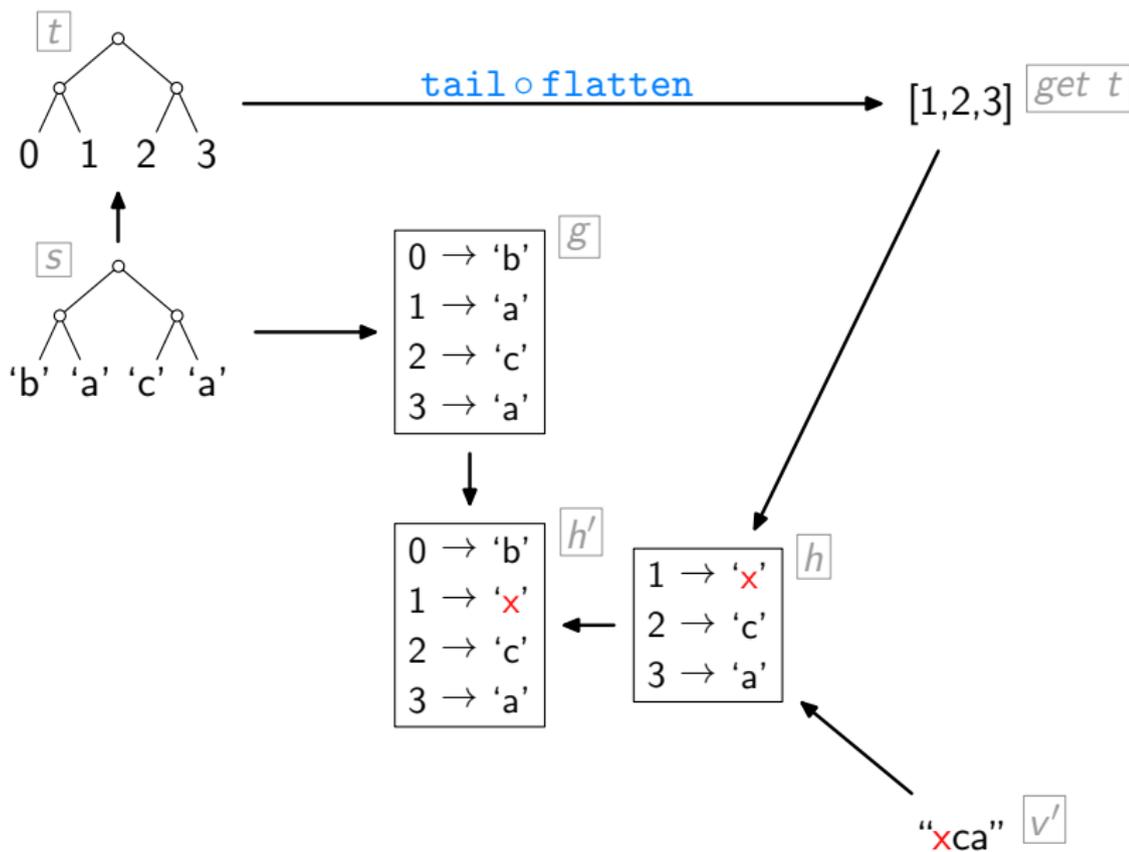


`"xca"`  $v'$

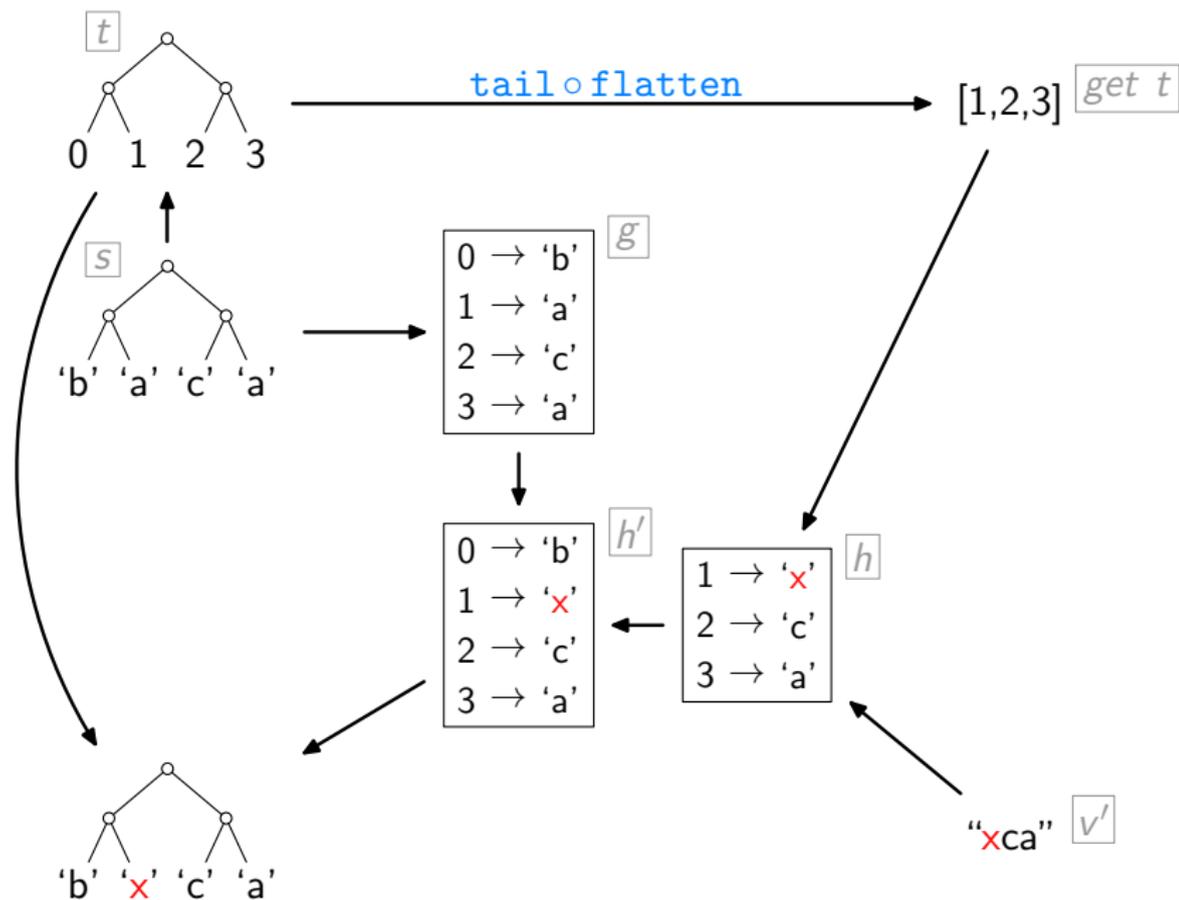
# Die resultierende Bidirektionalisierungsmethode in Aktion



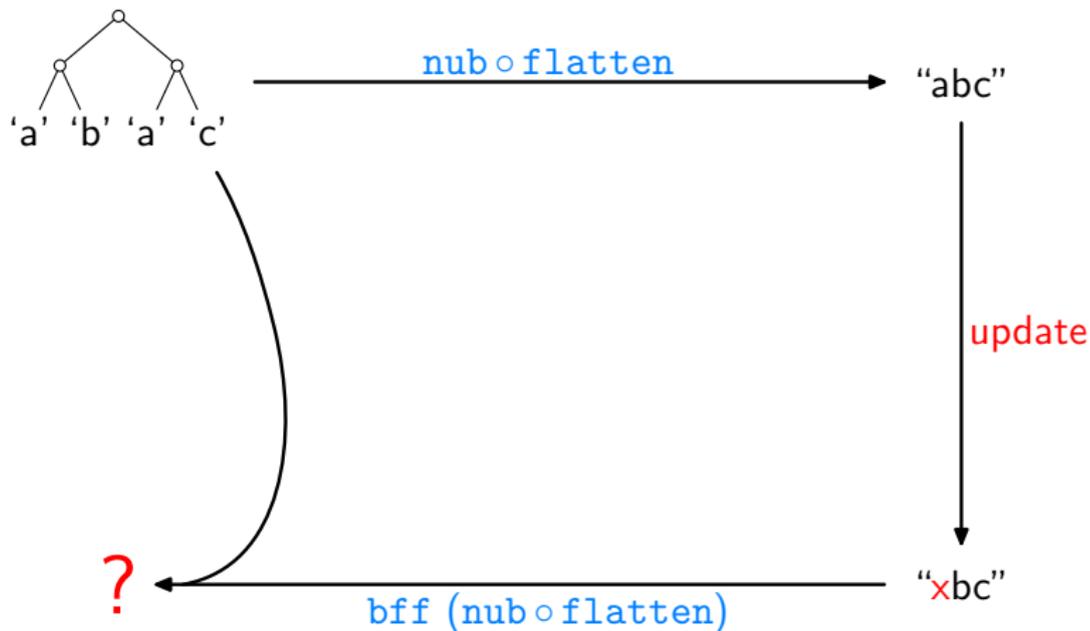
# Die resultierende Bidirektionalisierungsmethode in Aktion



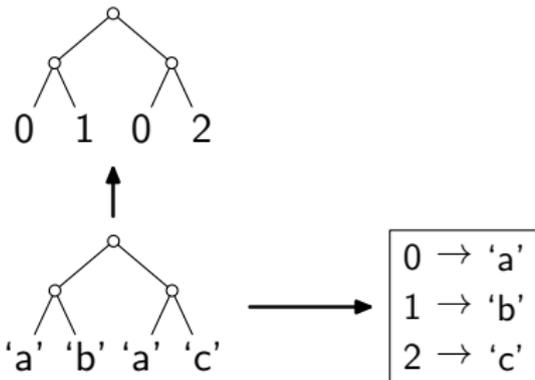
# Die resultierende Bidirektionalisierungs-Methode in Aktion



# Ein weiteres interessantes Beispiel (mit Typklasse Eq)

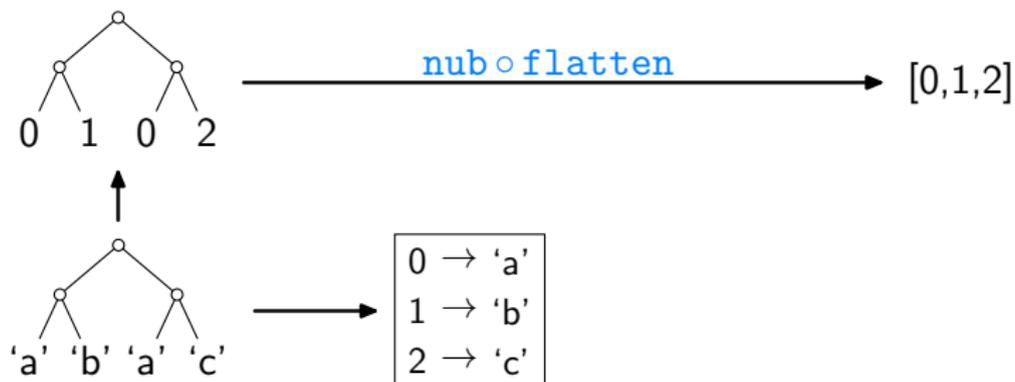


## Ein weiteres interessantes Beispiel (mit Typklasse Eq)



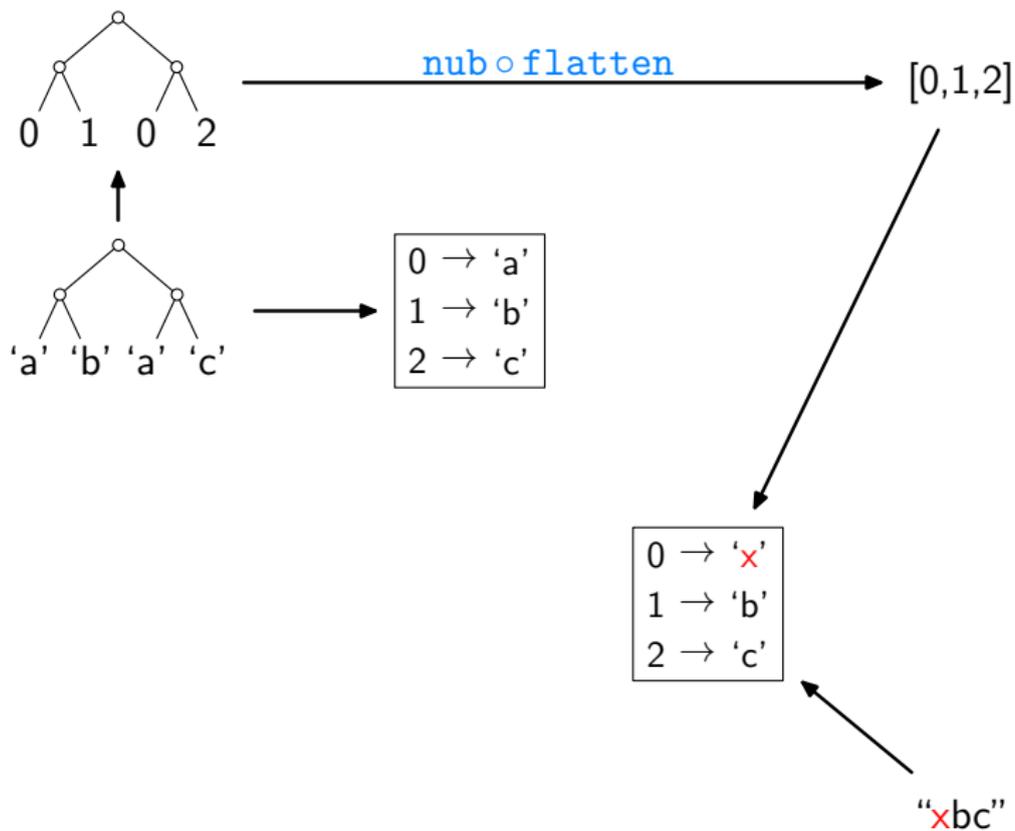
“x~~bc~~”

## Ein weiteres interessantes Beispiel (mit Typklasse Eq)

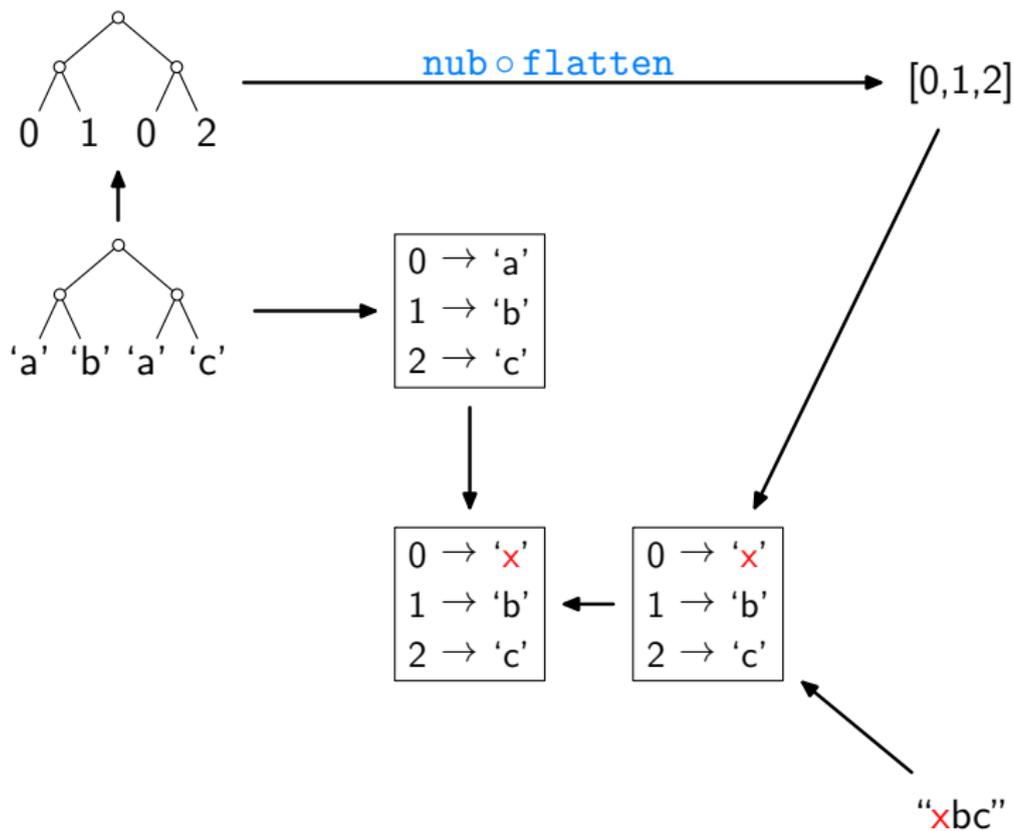


"x~~b~~c"

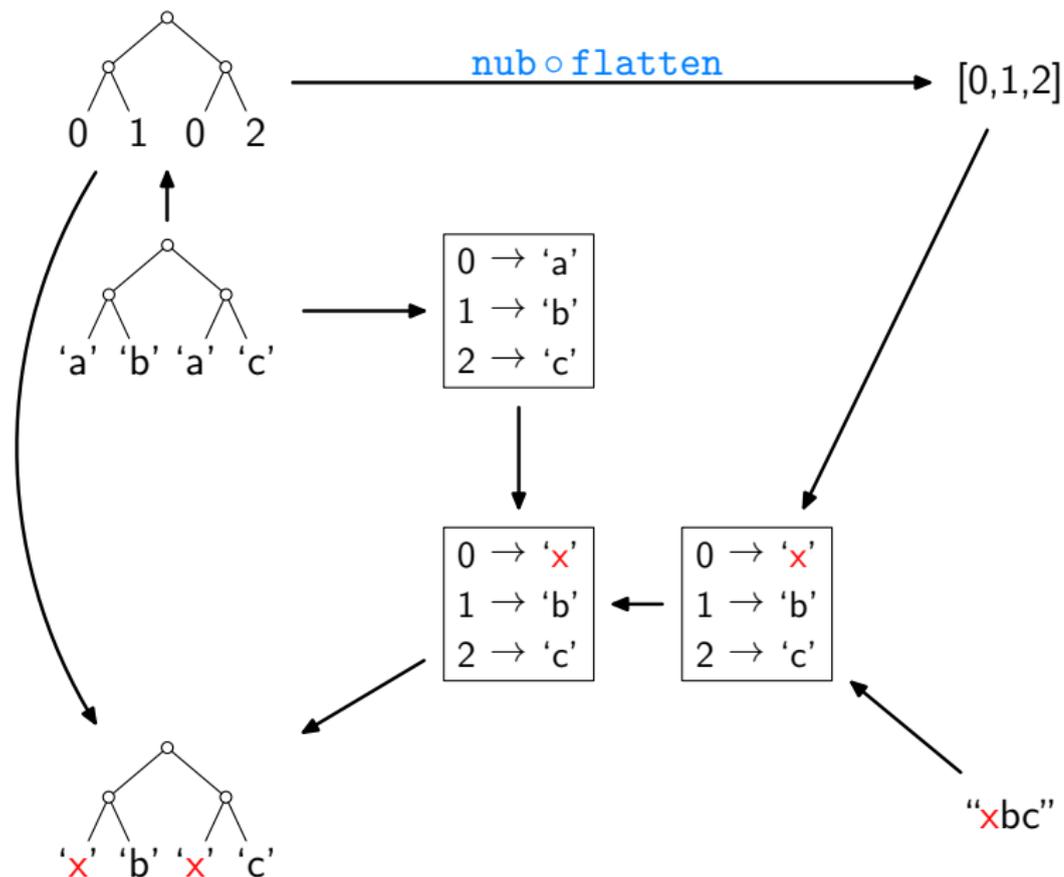
## Ein weiteres interessantes Beispiel (mit Typklasse Eq)



## Ein weiteres interessantes Beispiel (mit Typklasse Eq)



# Ein weiteres interessantes Beispiel (mit Typklasse Eq)



## Andere Erweiterung der Methode

Ein Hauptproblem:

- Shape-verändernde Updates führen zu Fehlschlag.

## Andere Erweiterung der Methode

Ein Hauptproblem:

- Shape-verändernde Updates führen zu Fehlschlag.
- Zum Beispiel, `bff tail "abcde" "xyz" ...`

## Andere Erweiterung der Methode

### Ein Hauptproblem:

- Shape-verändernde Updates führen zu Fehlschlag.
- Zum Beispiel, `bff tail "abcde" "xyz" ...`

### Weil:

- Unser Ansatz zur „Injektivierung“ von

$$\lambda s \rightarrow (\text{get } s, \text{compl } s)$$

war, mittels `compl` folgende Informationen festzuhalten:

1. Länge der ursprünglichen Liste
2. verworfene Listenelemente

## Andere Erweiterung der Methode

### Ein Hauptproblem:

- Shape-verändernde Updates führen zu Fehlschlag.
- Zum Beispiel, `bff tail "abcde" "xyz" ...`

### Weil:

- Unser Ansatz zur „Injektivierung“ von

$$\lambda s \rightarrow (\text{get } s, \text{compl } s)$$

war, mittels `compl` folgende Informationen festzuhalten:

1. Länge der ursprünglichen Liste
  2. verworfene Listenelemente
- Derart konservativ zu sein, ist oft nicht „nicht-injektiv“ genug.

## Andere Erweiterung der Methode

### Ein Hauptproblem:

- Shape-verändernde Updates führen zu Fehlschlag.
- Zum Beispiel, `buff tail` “abcde” “xyz” ...

### Weil:

- Unser Ansatz zur „Injektivierung“ von

$$\lambda s \rightarrow (\text{get } s, \text{compl } s)$$

war, mittels `compl` folgende Informationen festzuhalten:

1. Länge der ursprünglichen Liste
  2. verworfene Listenelemente
- Derart konservativ zu sein, ist oft nicht „nicht-injektiv“ genug.
  - Zum Beispiel:

`get = tail`  $\rightsquigarrow$  `put` “abcde” “xyz” schlägt fehl genau weil  
`compl` “abcde” = (5, [(0, 'a')])

## Annahme von Shape-Injektivität

Angenommen, es gäbe eine Funktion

`shapeInv :: Int → Int`

so dass, für jede ursprüngliche Liste `s`,

`length s = shapeInv (length (get s))`

## Annahme von Shape-Injektivität

Angenommen, es gäbe eine Funktion

`shapeInv :: Int → Int`

so dass, für jede ursprüngliche Liste `s`,

`length s = shapeInv (length (get s))`

Dann:

```
compl :: [α] → (Int, IntMap α)
compl s = let n = (length s) - 1
            t = [0..n]
            g = zip t s
            g' = filter (λ(i, _) → notElem i (get t)) g
            in (n + 1, g')
```

## Annahme von Shape-Injektivität

Angenommen, es gäbe eine Funktion

`shapeInv :: Int → Int`

so dass, für jede ursprüngliche Liste `s`,

`length s = shapeInv (length (get s))`

Dann:

```
compl :: [α] → IntMap α
compl s = let n = (length s) - 1
            t = [0..n]
            g = zip t s
            g' = filter (λ(i, _) → notElem i (get t)) g
            in g'
```

## Annahme von Shape-Injektivität

```
inv :: ([α], (Int, IntMap α)) → [α]
```

```
inv (v', (n + 1, g')) = let t = [0..n]
```

```
    h = assoc (get t) v'
```

```
    h' = h ++ g'
```

```
  in seq h (map (λi → fromJust (lookup i h')) t)
```

## Annahme von Shape-Injektivität

```
inv :: ([α],      IntMap α ) → [α]
inv (v',        g' ) = let n = (shapeInv (length v')) - 1
                        t  = [0..n]
                        h  = assoc (get t) v'
                        h' = h ++ g'
                    in seq h (map (λi → fromJust (lookup i h')) t)
```

## Annahme von Shape-Injektivität

```
inv :: ([α],      IntMap α ) → [α]
inv (v',        g' ) = let n = (shapeInv (length v')) - 1
                        t = [0..n]
                        h = assoc (get t) v'
                        h' = h ++ g'
                    in seq h (map (λi → fromJust (lookup i h')) t)
```

Aber wie `shapeInv` erhalten???

## Annahme von Shape-Injektivität

```
inv :: ([α],      IntMap α ) → [α]
inv (v',        g' ) = let n = (shapeInv (length v')) - 1
                        t = [0..n]
                        h = assoc (get t) v'
                        h' = h ++ g'
                    in seq h (map (λi → fromJust (lookup i h')) t)
```

Aber wie `shapeInv` erhalten???

Eine Möglichkeit: durch Nutzer angeben lassen.

## Annahme von Shape-Injektivität

```
inv :: ([α],      IntMap α ) → [α]
inv (v',        g' ) = let n = (shapeInv (length v')) - 1
                        t  = [0..n]
                        h  = assoc (get t) v'
                        h' = h ++ g'
                    in seq h (map (λi → fromJust (lookup i h')) t)
```

Aber wie `shapeInv` erhalten???

Eine Möglichkeit: durch Nutzer angeben lassen.

Eine andere Möglichkeit: statisch ermitteln (durch Typsystem?).

## Annahme von Shape-Injektivität

```
inv :: ([α],      IntMap α ) → [α]
inv (v',        g' ) = let n = (shapeInv (length v')) - 1
                        t  = [0..n]
                        h  = assoc (get t) v'
                        h' = h ++ g'
                    in seq h (map (λi → fromJust (lookup i h')) t)
```

Aber wie `shapeInv` erhalten???

Eine Möglichkeit: durch Nutzer angeben lassen.

Eine andere Möglichkeit: statisch ermitteln (durch Typsystem?).

Nur zum Experimentieren:

```
shapeInv :: Int → Int
```

```
shapeInv l = head [n + 1 | n ← [0..], (length (get [0..n])) == l]
```

## Noch nicht ganz geschafft

Funktioniert ganz gut in einigen Fällen:

`get = tail`  $\rightsquigarrow$  `put "abcde" "xyz" = "axyz"`, unter Verwendung von  
`compl "abcde" = [(0, 'a')]`

## Noch nicht ganz geschafft

Funktioniert ganz gut in einigen Fällen:

`get = tail`  $\rightsquigarrow$  `put "abcde" "xyz" = "axyz"`, unter Verwendung von  
`compl "abcde" = [(0, 'a')]`

Aber nicht so gut in anderen Fällen:

`get = init`  $\rightsquigarrow$  `put "abcde" "xyz"` schlägt fehl

## Noch nicht ganz geschafft

Funktioniert ganz gut in einigen Fällen:

`get = tail`  $\rightsquigarrow$  `put "abcde" "xyz" = "axyz"`, unter Verwendung von  
`compl "abcde" = [(0, 'a')]`

Aber nicht so gut in anderen Fällen:

`get = init`  $\rightsquigarrow$  `put "abcde" "xyz"` schlägt fehl, weil  
`compl "abcde" = [(4, 'e')]`

## Noch nicht ganz geschafft

Funktioniert ganz gut in einigen Fällen:

`get = tail`  $\rightsquigarrow$  `put "abcde" "xyz" = "axyz"`, unter Verwendung von  
`compl "abcde" = [(0, 'a')]`

Aber nicht so gut in anderen Fällen:

`get = init`  $\rightsquigarrow$  `put "abcde" "xyz"` schlägt fehl, weil  
`compl "abcde" = [(4, 'e')]`

Das Problem: durch Vorhalten der Indizes ist `compl` noch nicht „nicht-injektiv“ genug.

## Noch nicht ganz geschafft

Funktioniert ganz gut in einigen Fällen:

`get = tail`  $\rightsquigarrow$  `put "abcde" "xyz" = "axyz"`, unter Verwendung von  
`compl "abcde" = [(0, 'a')]`

Aber nicht so gut in anderen Fällen:

`get = init`  $\rightsquigarrow$  `put "abcde" "xyz"` schlägt fehl, weil  
`compl "abcde" = [(4, 'e')]`

Das Problem: durch Vorhalten der Indizes ist `compl` noch nicht „nicht-injektiv“ genug.

Beachte: selbst ohne diese Indizes wäre  $\lambda s \rightarrow (\text{get } s, \text{compl } s)$   
injektiv.

## Elimination von Indizes

```
compl :: [α] → [(Int, α)]  
compl s = let n = (length s) - 1  
           t = [0..n]  
           g = zip t s  
           g' = filter (λ(i, _) → notElem i (get t)) g  
         in g'
```

## Elimination von Indizes

```
compl :: [α] → [ α ]  
compl s = let n = (length s) - 1  
            t = [0..n]  
            g = zip t s  
            g' = filter (λ(i, _) → notElem i (get t)) g  
          in map snd g'
```

## Elimination von Indizes

```
compl :: [α] → [ α ]
compl s = let n = (length s) - 1
           t = [0..n]
           g = zip t s
           g' = filter (λ(i, _) → notElem i (get t)) g
           in map snd g'
```

```
inv :: ([α], [(Int, α)]) → [α]
inv (v', g') = let n = (shapeInv (length v')) - 1
                t = [0..n]
                h = assoc (get t) v'
                h' = h ++ g'
                in seq h (map (λi → fromJust (lookup i h')) t)
```

## Elimination von Indizes

```
compl :: [α] → [ α ]  
compl s = let n = (length s) - 1  
            t = [0..n]  
            g = zip t s  
            g' = filter (λ(i, _) → notElem i (get t)) g  
            in map snd g'
```

```
inv :: ([α], [ α ]) → [α]  
inv (v', c) = let n = (shapeInv (length v')) - 1  
               t = [0..n]  
               h = assoc (get t) v'  
               g' = zip (filter (λi → notElem i (get t)) t) c  
               h' = h ++ g'  
               in seq h (map (λi → fromJust (lookup i h')) t)
```

## Elimination von Indizes

```
compl :: [α] → [ α ]  
compl s = let n = (length s) - 1  
            t = [0..n]  
            g = zip t s  
            g' = filter (λ(i, _) → notElem i (get t)) g  
            in map snd g'
```

```
inv :: ([α], [ α ]) → [α]  
inv (v', c) = let n = (shapeInv (length v')) - 1  
               t = [0..n]  
               h = assoc (get t) v'  
               g' = zip (filter (λi → notElem i (get t)) t) c  
               h' = h ++ g'  
               in seq h (map (λi → fromJust (lookup i h')) t)
```

Nun:

```
get = init  ~>  put "abcde" "xyz" = "xyze"
```

## Weitere Beispiele

Sei `get = sieve` mit:

`sieve` ::  $[\alpha] \rightarrow [\alpha]$

`sieve` ( $a : b : cs$ ) =  $b : (\text{sieve } cs)$

`sieve` \_ = []

## Weitere Beispiele

Sei `get = sieve` mit:

```
sieve :: [α] → [α]
sieve (a : b : cs) = b : (sieve cs)
sieve _           = []
```

Dann:

```
put [1..8] [2, -4, 6, 8] = [1, 2, 3, -4, 5, 6, 7, 8]
```

## Weitere Beispiele

Sei `get = sieve` mit:

```
sieve :: [α] → [α]
sieve (a : b : cs) = b : (sieve cs)
sieve _           = []
```

Dann:

```
put [1..8] [2, -4, 6, 8]      = [1, 2, 3, -4, 5, 6, 7, 8]
put [1..8] [2, -4, 6]        = [1, 2, 3, -4, 5, 6]
```

## Weitere Beispiele

Sei `get = sieve` mit:

```
sieve :: [α] → [α]
sieve (a : b : cs) = b : (sieve cs)
sieve _           = []
```

Dann:

```
put [1..8] [2, -4, 6, 8]      = [1, 2, 3, -4, 5, 6, 7, 8]
put [1..8] [2, -4, 6]        = [1, 2, 3, -4, 5, 6]
put [1..8] [2, -4, 6, 8, 10, 12] = [1, 2, 3, -4, 5, 6, 7, 8, ⊥, 10, ⊥, 12]
```

## Weitere Beispiele

Sei `get = sieve` mit:

```
sieve :: [α] → [α]
sieve (a : b : cs) = b : (sieve cs)
sieve _           = []
```

Dann:

```
put [1..8] [2, -4, 6, 8]      = [1, 2, 3, -4, 5, 6, 7, 8]
put [1..8] [2, -4, 6]        = [1, 2, 3, -4, 5, 6]
put [1..8] [2, -4, 6, 8, 10, 12] = [1, 2, 3, -4, 5, 6, 7, 8, ⊥, 10, ⊥, 12]
```

Allerdings:

```
put [1..8] [0, 2, -4, 6, 8]  = [1, 0, 3, 2, 5, -4, 7, 6, ⊥, 8]
```

## Weitere Beispiele

Sei `get` = `sieve` mit:

```
sieve :: [α] → [α]
sieve (a : b : cs) = b : (sieve cs)
sieve _           = []
```

Dann:

```
put [1..8] [2, -4, 6, 8]      = [1, 2, 3, -4, 5, 6, 7, 8]
put [1..8] [2, -4, 6]        = [1, 2, 3, -4, 5, 6]
put [1..8] [2, -4, 6, 8, 10, 12] = [1, 2, 3, -4, 5, 6, 7, 8, ⊥, 10, ⊥, 12]
```

Allerdings:

```
put [1..8] [0, 2, -4, 6, 8] = [1, 0, 3, 2, 5, -4, 7, 6, ⊥, 8]
```

Wobei wir vielleicht lieber gehabt hätten:

```
put [1..8] [0, 2, -4, 6, 8] = [⊥, 0, 1, 2, 3, -4, 5, 6, 7, 8]
```

# Freie Theoreme und Anwendungen

Wie wir gesehen haben,

- schränken Typen das Verhalten von Programmen ein,

# Freie Theoreme und Anwendungen

Wie wir gesehen haben,

- schränken Typen das Verhalten von Programmen ein,
- führen so zu interessanten Erkenntnissen über Programme,

# Freie Theoreme und Anwendungen

Wie wir gesehen haben,

- schränken Typen das Verhalten von Programmen ein,
- führen so zu interessanten Erkenntnissen über Programme,
- gut kombinierbar mit algebraischen Techniken und gleichungsbasiertem Schließen.

# Freie Theoreme und Anwendungen

Wie wir gesehen haben,

- schränken Typen das Verhalten von Programmen ein,
- führen so zu interessanten Erkenntnissen über Programme,
- gut kombinierbar mit algebraischen Techniken und gleichungsbasiertem Schließen.

Anwendungsgebiete umfassen:

- Programmoptimierungen
- spezifischere Domains

# Freie Theoreme und Anwendungen

Wie wir gesehen haben,

- schränken Typen das Verhalten von Programmen ein,
- führen so zu interessanten Erkenntnissen über Programme,
- gut kombinierbar mit algebraischen Techniken und gleichungsbasiertem Schließen.

Anwendungsgebiete umfassen:

- Programmoptimierungen
- spezifischere Domains

Aber:

- Wie ist das Ganze formal garantiert?

# Freie Theoreme und Anwendungen

Wie wir gesehen haben,

- schränken Typen das Verhalten von Programmen ein,
- führen so zu interessanten Erkenntnissen über Programme,
- gut kombinierbar mit algebraischen Techniken und gleichungsbasiertem Schließen.

Anwendungsgebiete umfassen:

- Programmoptimierungen
- spezifischere Domains

Aber:

- Wie ist das Ganze formal garantiert?
- Decken wir wirklich die ganze Sprache Haskell ab?

# Freie Theoreme und Anwendungen

Wie wir gesehen haben,

- schränken Typen das Verhalten von Programmen ein,
- führen so zu interessanten Erkenntnissen über Programme,
- gut kombinierbar mit algebraischen Techniken und gleichungsbasiertem Schließen.

Anwendungsgebiete umfassen:

- Programmoptimierungen
- spezifischere Domains

Aber:

- Wie ist das Ganze formal garantiert?
- Decken wir wirklich die ganze Sprache Haskell ab?
- Ist noch mehr drin?

## 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.
- 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?

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

## Ein kompositioneller Ansatz

Frage: Welche  $g$  haben einen gewissen Typ  $\tau$ ?

Ansatz: Denotationen  $\llbracket \tau \rrbracket$  von Typen als Mengen angeben.

$$\begin{aligned}\llbracket \text{Bool} \rrbracket &= \{\text{True}, \text{False}\} \\ \llbracket \text{Int} \rrbracket &= \{\dots, -2, -1, 0, 1, 2, \dots\} \\ \llbracket (\tau_1, \tau_2) \rrbracket &= \llbracket \tau_1 \rrbracket \times \llbracket \tau_2 \rrbracket \\ \llbracket [\tau] \rrbracket &= \{[x_1, \dots, x_n] \mid n \geq 0, x_i \in \llbracket \tau \rrbracket\} \\ \llbracket \tau_1 \rightarrow \tau_2 \rrbracket &= \{f : \llbracket \tau_1 \rrbracket \rightarrow \llbracket \tau_2 \rrbracket\} \\ \llbracket \forall \alpha. \tau \rrbracket &= ?\end{aligned}$$

- $g \in \llbracket \forall \alpha. \tau \rrbracket$  müsste eine ganze „Familie“ von Werten sein: für jeden Typ  $\tau'$ , eine Instanz mit Typ  $\tau[\tau'/\alpha]$ .
- $\llbracket \forall \alpha. \tau \rrbracket = \{g \mid \forall \tau'. g_{\tau'} \in \llbracket \tau[\tau'/\alpha] \rrbracket\} ?$
- Aber das schließt „ad-hoc-polymorphe“ Funktionen ein!

## Idee [Reynolds 1983]

Beliebige Relationen benutzen, um Instanzen zu verknüpfen!

Im Beispiel ( $g :: \forall \alpha. (\alpha, \alpha) \rightarrow \alpha$ ):

- Wähle eine Relation  $\mathcal{R} \subseteq \llbracket \text{Bool} \rrbracket \times \llbracket \text{Int} \rrbracket$ .
- Nenne  $(x_1, x_2) \in \llbracket \text{Bool} \rrbracket \times \llbracket \text{Bool} \rrbracket$  und  $(y_1, y_2) \in \llbracket \text{Int} \rrbracket \times \llbracket \text{Int} \rrbracket$  verwandt, wenn  $(x_1, y_1) \in \mathcal{R}$  und  $(x_2, y_2) \in \mathcal{R}$ .
- Nenne  $f_1 : \llbracket \text{Bool} \rrbracket \times \llbracket \text{Bool} \rrbracket \rightarrow \llbracket \text{Bool} \rrbracket, f_2 : \llbracket \text{Int} \rrbracket \times \llbracket \text{Int} \rrbracket \rightarrow \llbracket \text{Int} \rrbracket$  verwandt, wenn verwandte Eingaben zu verwandten Ausgaben führen.
- Dann sind  $g_{\text{Bool}}$  und  $g_{\text{Int}}$  mit

$$\begin{aligned}g_{\text{Bool}}(x, y) &= \neg x \quad \text{und} \\g_{\text{Int}}(x, y) &= y + 1\end{aligned}$$

nicht verwandt bei, zum Beispiel, Wahl von  $\mathcal{R} = \{(\text{True}, 1)\}$ .

Reynolds:  $g \in \llbracket \forall \alpha. \tau \rrbracket$  genau dann wenn für alle  $\tau_1, \tau_2$  und  $\mathcal{R} \subseteq \llbracket \tau_1 \rrbracket \times \llbracket \tau_2 \rrbracket$  gilt, dass  $g_{\tau_1}$  und  $g_{\tau_2}$  verwandt per „Fortsetzung“ von  $\mathcal{R}$  bezüglich  $\tau$  sind

## Freie Theoreme, allgemein

Zur Interpretation von Typen als Relationen:

1. Ersetze (Quantifizierung über) Typvariablen durch (Quantifizierung über) Relationsvariablen.
2. Ersetze Teiltypen ohne jeglichen Polymorphismus durch Identitätsrelationen.
3. Verwende folgende Regeln:

$$(\mathcal{R}, \mathcal{S}) = \{((x_1, x_2), (y_1, y_2)) \mid (x_1, y_1) \in \mathcal{R}, (x_2, y_2) \in \mathcal{S}\}$$

$$[\mathcal{R}] = \{([x_1, \dots, x_n], [y_1, \dots, y_n]) \mid n \geq 0, (x_i, y_i) \in \mathcal{R}\}$$

$$\mathcal{R} \rightarrow \mathcal{S} = \{(f_1, f_2) \mid \forall (a_1, a_2) \in \mathcal{R}. (f_1 a_1, f_2 a_2) \in \mathcal{S}\}$$

$$\forall \mathcal{R}. \mathcal{F}(\mathcal{R}) = \{(u, v) \mid \forall \tau_1, \tau_2, \mathcal{R} \subseteq \tau_1 \times \tau_2. (u_{\tau_1}, v_{\tau_2}) \in \mathcal{F}(\mathcal{R})\}$$

Dann gilt für jedes  $g :: \tau$ , dass das Paar  $(g, g)$  in der Interpretation von  $\tau$  als Relation enthalten 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

# Deskriptive Programmierung

SS 2009



## 3.3 Theoretische Grundlagen der funktionalen Programmierung



Der Lambda-Kalkül besteht aus *zwei* Bausteinen:

- Funktionsabstraktion

$\lambda x . A$  definiert eine (anonyme) Funktion, die ein  $x$  als Parameter bekommt und einen Ausdruck  $A$  als Funktionskörper hat (in dem wiederum  $x$  vorkommen kann aber nicht muss).

- Funktionsapplikation

$F A$  bedeutet, dass die Funktion  $F$  auf den Ausdruck  $A$  angewandt wird

Bemerkung: Die folgende Darstellung des Lambda-Kalküls orientiert sich an der ‚online‘-Einführung von Fabian Nilius.

- Identität:

$$\lambda x . x$$

- Eine Funktion, die jedes Argument auf die Identitätsfunktion abbildet:

$$\lambda y . (\lambda x . x)$$

- Die Identitätsfunktion angewendet auf sich selbst:

$$(\lambda x . x) (\lambda y . y) \rightarrow (\lambda y . y)$$

## Beispiele für Funktionen (3)

- Eine komplexere Funktion:

$\lambda f. (\lambda x. f (f x))$

- Ein damit konstruierter Ausdruck:

$\lambda f. (\lambda x. f (f x)) \ u \ v$

$\equiv \lambda f. (\lambda x. f (f x)) \ u \ v$

$\rightarrow (\lambda x. u (u x)) \ v$

$\rightarrow u (u v)$

Die Variable  $f$  wird auch in den inneren Ausdrücken ersetzt.

Diese Funktion wendet also eine Funktion zweimal auf ein Argument an!

- Es gibt zwei grundlegende Ableitungsregeln in diesem Kalkül:
  - **$\alpha$ -Konversion** (gestattet den Austausch von Variablennamen)

$$(\lambda x . A) \leftrightarrow (\lambda y . A'),$$

wobei in  $A'$  alle Vorkommnisse von  $x$  durch  $y$  ersetzt wurden

- **$\beta$ -Konversion** (Applikation einer Funktion)

$$(\lambda x . A) (B) \leftrightarrow A',$$

wobei in  $A'$  alle Vorkommnisse von  $x$  durch  $B$  ersetzt wurden

- Die  $\beta$ -Konversion wurde bereits bei der Anwendung der Identitätsfunktion auf sich selbst verwendet:

$$(\lambda x . x) (\lambda y . y) \xrightarrow{\beta} (\lambda y . y)$$

## Ableitungsregeln (2)

- Wir haben bisher nicht definiert, wie bei einer Funktionsanwendung die Variablen ersetzt werden sollen.
- **Sichtbarkeit:** Schachteln wir F-Definitionen sind die weiter außen definierten Variablennamen innen sichtbar.
- Die  $\beta$ -Konversion drückt das dadurch aus, dass die Argumente für alle Vorkommen der Variable ersetzt werden:

$$(\lambda f . (\lambda x . f (f x))) u$$

- Allerdings sind nicht alle Variablen beim Ersetzungsprozess **frei**, da sie durch eine weitere F-Abstraktion gebunden sein können (hier z.B. das  $x$ ).

- Eine Variable  $x$  ist in einem Ausdruck **frei** bzw. **gebunden**:

$x$  ist frei in  $x$  (aber nicht in einer anderen Variable)

$x$  ist frei in  $E F \iff x$  frei in  $E$  **oder**  $x$  frei in  $F$

$x$  ist frei in  $\lambda y. E \iff x \neq y$  **und**  $x$  frei in  $E$

- Beim Ersetzen von Variablen werden nur die **freien** Vorkommen herangezogen:

$(\lambda f. f(\lambda f. f)) A$

Richtig:

$\rightarrow A(\lambda f. f)$

Falsch:

$\rightarrow A(\lambda f. A)$

$\rightarrow A(\lambda A. A)$

- Wenn Variablennamen mehrfach und sowohl frei als auch gebunden auftreten, können weitere Namenskonflikte auftreten:

$\lambda x . (\lambda y . (\lambda x . y) x)$

Setzen wir einfach dieses x für das y ein, so bezieht sich das x plötzlich auf die innerste Funktionsdefinition!

- Um diese Form von Namenskonflikten zu vermeiden, verwenden wir die  $\alpha$ -Konversion zur Umbenennung (hier x zu x'):

$$\lambda x . (\lambda y . (\lambda x' . y) x)$$
$$\rightarrow \lambda x . (\lambda x' . x)$$

- Der **Kern** des  $\lambda$ -Kalküls bietet keine Konstanten (wie 1, "foo", True) oder vordefinierte Funktionen (wie +,  $\times$ , if).
- Im folgenden werden wir einen um Konstanten und vordef. Funktionen **erweiterten Kalkül** verwenden (legitimiert durch die vorgestellten Funktionssimulationen).

### $\beta$ -Konversion einiger vordefinierter Funktionen:

$$\begin{aligned} (+ x y) &\rightarrow x \oplus y \\ (\times x y) &\rightarrow x \otimes y \\ (\text{if True } e f) &\rightarrow e \\ (\text{if False } e f) &\rightarrow f \\ (\text{and False } e) &\rightarrow \text{False} \\ (\text{and True } e) &\rightarrow e \end{aligned}$$

(Operationen in  $\circ$  direkt auf der Zielmaschine ausführbar, wenn die Argumente  $x$  und  $y$  zuvor bis zur Normalform reduziert wurden.)

## Beispiele für das erweiterte $\lambda$ -Kalkül (1)

- Funktion, die ihr Argument inkrementiert:

$$\lambda x . (+ x 1)$$

- Maximumsfunktion:

$$\lambda x . (\lambda y . (\text{if } (< x y) y x))$$

- $\beta$ -Konversion mit freien und gebundenen Variablen:

$(\lambda x . (\lambda x . (+ (\times x 2))) x 3) 9$	$\xrightarrow{\beta}$	$(\lambda x . (+ (\times x 2))) 9 3$
	$\xrightarrow{\beta}$	$+ (\times 9 2) 3$
	$\rightarrow$	$+ (9 \otimes 2) 3 \rightarrow + 18 3 \rightarrow \dots$

Einfach **typisierter**  $\lambda$ -Kalkül:

- bisherige Syntax bleibt erhalten
- nur noch typisierbare Terme:
  - Typen:  $\tau := \text{bool} \mid \text{nat} \mid \dots \mid \alpha \mid \beta \mid \dots \mid \tau \rightarrow \tau$
  - **Typisierungskontext**:  $\Gamma: V \rightarrow \tau$  (partielle Funktion über Variablenmenge  $V$ )
  - **Typisierungsregeln**:

$$\frac{\Gamma(x) \text{ ist definiert}}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{\Gamma \vdash E_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash E_2 : \tau_1}{\Gamma \vdash (E_1 E_2) : \tau_2}$$

$$\frac{\Gamma[x:\tau] \vdash E : \tau'}{\Gamma \vdash (\lambda x.E) : \tau \rightarrow \tau'}$$

- Terme z.B. der Form  $(E E)$  sind nicht typisierbar!

# Polymorpher Lambda-Kalkül [Girard 1972, Reynolds 1974]

Typen:  $\tau := \alpha \mid \tau \rightarrow \tau \mid \forall \alpha. \tau$

Terme:  $t := x \mid \lambda x : \tau. t \mid t t \mid \Lambda \alpha. t \mid t \tau$

# Polymorpher Lambda-Kalkül [Girard 1972, Reynolds 1974]

Typen:  $\tau := \alpha \mid \tau \rightarrow \tau \mid \forall \alpha. \tau$

Terme:  $t := x \mid \lambda x : \tau. t \mid t t \mid \Lambda \alpha. t \mid t \tau$

$\Gamma, x : \tau \vdash x : \tau$

## Polymorpher Lambda-Kalkül [Girard 1972, Reynolds 1974]

Typen:  $\tau := \alpha \mid \tau \rightarrow \tau \mid \forall \alpha. \tau$

Terme:  $t := x \mid \lambda x : \tau. t \mid t t \mid \Lambda \alpha. t \mid t \tau$

$\Gamma, x : \tau \vdash x : \tau$

$$\frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash (\lambda x : \tau_1. t) : \tau_1 \rightarrow \tau_2}$$

# Polymorpher Lambda-Kalkül [Girard 1972, Reynolds 1974]

Typen:  $\tau := \alpha \mid \tau \rightarrow \tau \mid \forall \alpha. \tau$

Terme:  $t := x \mid \lambda x : \tau. t \mid t t \mid \Lambda \alpha. t \mid t \tau$

$\Gamma, x : \tau \vdash x : \tau$

$$\frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash (\lambda x : \tau_1. t) : \tau_1 \rightarrow \tau_2}$$
$$\frac{\Gamma \vdash t : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash u : \tau_1}{\Gamma \vdash (t u) : \tau_2}$$

# Polymorpher Lambda-Kalkül [Girard 1972, Reynolds 1974]

Typen:  $\tau := \alpha \mid \tau \rightarrow \tau \mid \forall \alpha. \tau$

Terme:  $t := x \mid \lambda x : \tau. t \mid t t \mid \Lambda \alpha. t \mid t \tau$

$\Gamma, x : \tau \vdash x : \tau$

$$\frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash (\lambda x : \tau_1. t) : \tau_1 \rightarrow \tau_2}$$
$$\frac{\Gamma \vdash t : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash u : \tau_1}{\Gamma \vdash (t u) : \tau_2}$$
$$\frac{\alpha, \Gamma \vdash t : \tau}{\Gamma \vdash (\Lambda \alpha. t) : \forall \alpha. \tau}$$

# Polymorpher Lambda-Kalkül [Girard 1972, Reynolds 1974]

Typen:  $\tau := \alpha \mid \tau \rightarrow \tau \mid \forall \alpha. \tau$

Terme:  $t := x \mid \lambda x : \tau. t \mid t t \mid \Lambda \alpha. t \mid t \tau$

$\Gamma, x : \tau \vdash x : \tau$

$$\frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash (\lambda x : \tau_1. t) : \tau_1 \rightarrow \tau_2}$$
$$\frac{\Gamma \vdash t : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash u : \tau_1}{\Gamma \vdash (t u) : \tau_2}$$
$$\frac{\alpha, \Gamma \vdash t : \tau}{\Gamma \vdash (\Lambda \alpha. t) : \forall \alpha. \tau}$$
$$\frac{\Gamma \vdash t : \forall \alpha. \tau}{\Gamma \vdash (t \tau') : \tau[\tau'/\alpha]}$$

# Polymorpher Lambda-Kalkül [Girard 1972, Reynolds 1974]

Typen:  $\tau := \alpha \mid \tau \rightarrow \tau \mid \forall \alpha. \tau$

Terme:  $t := x \mid \lambda x : \tau. t \mid t t \mid \Lambda \alpha. t \mid t \tau$

$$\Gamma, x : \tau \vdash x : \tau \qquad \llbracket x \rrbracket_{\theta, \sigma} = \sigma(x)$$

$$\frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash (\lambda x : \tau_1. t) : \tau_1 \rightarrow \tau_2} \qquad \llbracket \lambda x : \tau_1. t \rrbracket_{\theta, \sigma} a = \llbracket t \rrbracket_{\theta, \sigma[x \mapsto a]}$$

$$\frac{\Gamma \vdash t : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash u : \tau_1}{\Gamma \vdash (t u) : \tau_2} \qquad \llbracket t u \rrbracket_{\theta, \sigma} = \llbracket t \rrbracket_{\theta, \sigma} \llbracket u \rrbracket_{\theta, \sigma}$$

$$\frac{\alpha, \Gamma \vdash t : \tau}{\Gamma \vdash (\Lambda \alpha. t) : \forall \alpha. \tau} \qquad \llbracket \Lambda \alpha. t \rrbracket_{\theta, \sigma} S = \llbracket t \rrbracket_{\theta[\alpha \mapsto S], \sigma}$$

$$\frac{\Gamma \vdash t : \forall \alpha. \tau}{\Gamma \vdash (t \tau') : \tau[\tau'/\alpha]} \qquad \llbracket t \tau' \rrbracket_{\theta, \sigma} = \llbracket t \rrbracket_{\theta, \sigma} \llbracket \tau' \rrbracket_{\theta}$$

## Das Parametritäts-Theorem [Reynolds 1983, Wadler 1989]

Gegeben  $\tau$  und Environments  $\theta_1, \theta_2, \rho$  mit  $\rho(\alpha) \subseteq \theta_1(\alpha) \times \theta_2(\alpha)$ ,  
definiere  $\Delta_{\tau, \rho} \subseteq \llbracket \tau \rrbracket_{\theta_1} \times \llbracket \tau \rrbracket_{\theta_2}$  wie folgt:

## Das Parametritäts-Theorem [Reynolds 1983, Wadler 1989]

Gegeben  $\tau$  und Environments  $\theta_1, \theta_2, \rho$  mit  $\rho(\alpha) \subseteq \theta_1(\alpha) \times \theta_2(\alpha)$ ,  
definiere  $\Delta_{\tau, \rho} \subseteq \llbracket \tau \rrbracket_{\theta_1} \times \llbracket \tau \rrbracket_{\theta_2}$  wie folgt:

$$\Delta_{\alpha, \rho} = \rho(\alpha)$$

## Das Parametritäts-Theorem [Reynolds 1983, Wadler 1989]

Gegeben  $\tau$  und Environments  $\theta_1, \theta_2, \rho$  mit  $\rho(\alpha) \subseteq \theta_1(\alpha) \times \theta_2(\alpha)$ ,  
definiere  $\Delta_{\tau, \rho} \subseteq \llbracket \tau \rrbracket_{\theta_1} \times \llbracket \tau \rrbracket_{\theta_2}$  wie folgt:

$$\Delta_{\alpha, \rho} = \rho(\alpha)$$

$$\Delta_{\tau_1 \rightarrow \tau_2, \rho} = \{(f_1, f_2) \mid \forall (a_1, a_2) \in \Delta_{\tau_1, \rho}. (f_1 a_1, f_2 a_2) \in \Delta_{\tau_2, \rho}\}$$

## Das Parametritäts-Theorem [Reynolds 1983, Wadler 1989]

Gegeben  $\tau$  und Environments  $\theta_1, \theta_2, \rho$  mit  $\rho(\alpha) \subseteq \theta_1(\alpha) \times \theta_2(\alpha)$ ,  
definiere  $\Delta_{\tau, \rho} \subseteq \llbracket \tau \rrbracket_{\theta_1} \times \llbracket \tau \rrbracket_{\theta_2}$  wie folgt:

$$\Delta_{\alpha, \rho} = \rho(\alpha)$$

$$\Delta_{\tau_1 \rightarrow \tau_2, \rho} = \{(f_1, f_2) \mid \forall (a_1, a_2) \in \Delta_{\tau_1, \rho}. (f_1 a_1, f_2 a_2) \in \Delta_{\tau_2, \rho}\}$$

$$\Delta_{\forall \alpha. \tau, \rho} = \{(g_1, g_2) \mid \forall \mathcal{R} \subseteq S_1 \times S_2. (g_1 S_1, g_2 S_2) \in \Delta_{\tau, \rho[\alpha \mapsto \mathcal{R}]}\}$$

## Das Parametritäts-Theorem [Reynolds 1983, Wadler 1989]

Gegeben  $\tau$  und Environments  $\theta_1, \theta_2, \rho$  mit  $\rho(\alpha) \subseteq \theta_1(\alpha) \times \theta_2(\alpha)$ ,  
definiere  $\Delta_{\tau, \rho} \subseteq \llbracket \tau \rrbracket_{\theta_1} \times \llbracket \tau \rrbracket_{\theta_2}$  wie folgt:

$$\Delta_{\alpha, \rho} = \rho(\alpha)$$

$$\Delta_{\tau_1 \rightarrow \tau_2, \rho} = \{(f_1, f_2) \mid \forall (a_1, a_2) \in \Delta_{\tau_1, \rho}. (f_1 a_1, f_2 a_2) \in \Delta_{\tau_2, \rho}\}$$

$$\Delta_{\forall \alpha. \tau, \rho} = \{(g_1, g_2) \mid \forall \mathcal{R} \subseteq S_1 \times S_2. (g_1 S_1, g_2 S_2) \in \Delta_{\tau, \rho[\alpha \mapsto \mathcal{R}]}\}$$

Dann gilt für jeden geschlossenen Term  $t$  geschlossenen Typs  $\tau$ :

$$(\llbracket t \rrbracket_{\emptyset, \emptyset}, \llbracket t \rrbracket_{\emptyset, \emptyset}) \in \Delta_{\tau, \emptyset}.$$