

## Problem: Zwischenergebnisse

Ein Beispiel:

```
filter ((== 1) ◦ ('mod' 3)) (map (^2) [1..5])
```

## Problem: Zwischenergebnisse

Ein Beispiel:

```
filter ((== 1) ◦ ('mod' 3)) (map (^2) [1..5])  
= filter ((== 1) ◦ ('mod' 3)) (1 : (map (^2) [2..5]))
```

## Problem: Zwischenergebnisse

Ein Beispiel:

```
filter ((== 1) ◦ ('mod' 3)) (map (^2) [1..5])  
= filter ((== 1) ◦ ('mod' 3)) (1 : (map (^2) [2..5]))  
= 1 : (filter ((== 1) ◦ ('mod' 3)) (map (^2) [2..5]))
```

## Problem: Zwischenergebnisse

Ein Beispiel:

```
filter ((== 1) ◦ ('mod' 3)) (map (^2) [1..5])  
= filter ((== 1) ◦ ('mod' 3)) (1 : (map (^2) [2..5]))  
= 1 : (filter ((== 1) ◦ ('mod' 3)) (map (^2) [2..5]))  
= 1 : (filter ((== 1) ◦ ('mod' 3)) (4 : (map (^2) [3..5])))
```

## Problem: Zwischenergebnisse

Ein Beispiel:

```
filter ((== 1) ◦ ('mod' 3)) (map (^2) [1..5])  
= filter ((== 1) ◦ ('mod' 3)) (1 : (map (^2) [2..5]))  
= 1 : (filter ((== 1) ◦ ('mod' 3)) (map (^2) [2..5]))  
= 1 : (filter ((== 1) ◦ ('mod' 3)) (4 : (map (^2) [3..5])))  
= 1 : 4 : (filter ((== 1) ◦ ('mod' 3)) (map (^2) [3..5]))
```

## Problem: Zwischenergebnisse

Ein Beispiel:

```
filter ((== 1) ◦ ('mod' 3)) (map (^2) [1..5])
= filter ((== 1) ◦ ('mod' 3)) (1 : (map (^2) [2..5]))
= 1 : (filter ((== 1) ◦ ('mod' 3)) (map (^2) [2..5]))
= 1 : (filter ((== 1) ◦ ('mod' 3)) (4 : (map (^2) [3..5])))
= 1 : 4 : (filter ((== 1) ◦ ('mod' 3)) (map (^2) [3..5]))
= 1 : 4 : (filter ((== 1) ◦ ('mod' 3)) (9 : (map (^2) [4, 5])))
```

## Problem: Zwischenergebnisse

Ein Beispiel:

```
filter ((== 1) ◦ ('mod' 3)) (map (^2) [1..5])
= filter ((== 1) ◦ ('mod' 3)) (1 : (map (^2) [2..5]))
= 1 : (filter ((== 1) ◦ ('mod' 3)) (map (^2) [2..5]))
= 1 : (filter ((== 1) ◦ ('mod' 3)) (4 : (map (^2) [3..5])))
= 1 : 4 : (filter ((== 1) ◦ ('mod' 3)) (map (^2) [3..5]))
= 1 : 4 : (filter ((== 1) ◦ ('mod' 3)) (9 : (map (^2) [4, 5])))
= 1 : 4 : (filter ((== 1) ◦ ('mod' 3)) (map (^2) [4, 5]))
```

## Problem: Zwischenergebnisse

Ein Beispiel:

```
filter ((== 1) ◦ ('mod' 3)) (map (^2) [1..5])
= filter ((== 1) ◦ ('mod' 3)) (1 : (map (^2) [2..5]))
= 1 : (filter ((== 1) ◦ ('mod' 3)) (map (^2) [2..5]))
= 1 : (filter ((== 1) ◦ ('mod' 3)) (4 : (map (^2) [3..5])))
= 1 : 4 : (filter ((== 1) ◦ ('mod' 3)) (map (^2) [3..5]))
= 1 : 4 : (filter ((== 1) ◦ ('mod' 3)) (9 : (map (^2) [4, 5])))
= 1 : 4 : (filter ((== 1) ◦ ('mod' 3)) (map (^2) [4, 5]))
= 1 : 4 : (filter ((== 1) ◦ ('mod' 3)) (16 : (map (^2) [5])))
```



## Problem: Zwischenergebnisse

Ein Beispiel:

```
filter ((== 1) ◦ ('mod' 3)) (map (^2) [1..5])
= filter ((== 1) ◦ ('mod' 3)) (1 : (map (^2) [2..5]))
= 1 : (filter ((== 1) ◦ ('mod' 3)) (map (^2) [2..5]))
= 1 : (filter ((== 1) ◦ ('mod' 3)) (4 : (map (^2) [3..5])))
= 1 : 4 : (filter ((== 1) ◦ ('mod' 3)) (map (^2) [3..5]))
= 1 : 4 : (filter ((== 1) ◦ ('mod' 3)) (9 : (map (^2) [4, 5])))
= 1 : 4 : (filter ((== 1) ◦ ('mod' 3)) (map (^2) [4, 5]))
= 1 : 4 : (filter ((== 1) ◦ ('mod' 3)) (16 : (map (^2) [5])))
= 1 : 4 : 16 : (filter ((== 1) ◦ ('mod' 3)) (map (^2) [5]))
```

## Problem: Zwischenergebnisse

Ein Beispiel:

```
filter ((== 1) ◦ ('mod' 3)) (map (^2) [1..5])
= filter ((== 1) ◦ ('mod' 3)) (1 : (map (^2) [2..5]))
= 1 : (filter ((== 1) ◦ ('mod' 3)) (map (^2) [2..5]))
= 1 : (filter ((== 1) ◦ ('mod' 3)) (4 : (map (^2) [3..5])))
= 1 : 4 : (filter ((== 1) ◦ ('mod' 3)) (map (^2) [3..5]))
= 1 : 4 : (filter ((== 1) ◦ ('mod' 3)) (9 : (map (^2) [4, 5])))
= 1 : 4 : (filter ((== 1) ◦ ('mod' 3)) (map (^2) [4, 5]))
= 1 : 4 : (filter ((== 1) ◦ ('mod' 3)) (16 : (map (^2) [5])))
= 1 : 4 : 16 : (filter ((== 1) ◦ ('mod' 3)) (map (^2) [5]))
= 1 : 4 : 16 : (filter ((== 1) ◦ ('mod' 3)) (25 : (map (^2) [])))
```

## Problem: Zwischenergebnisse

Ein Beispiel:

```
filter ((== 1) ◦ ('mod' 3)) (map (^2) [1..5])
= filter ((== 1) ◦ ('mod' 3)) (1 : (map (^2) [2..5]))
= 1 : (filter ((== 1) ◦ ('mod' 3)) (map (^2) [2..5]))
= 1 : (filter ((== 1) ◦ ('mod' 3)) (4 : (map (^2) [3..5])))
= 1 : 4 : (filter ((== 1) ◦ ('mod' 3)) (map (^2) [3..5]))
= 1 : 4 : (filter ((== 1) ◦ ('mod' 3)) (9 : (map (^2) [4, 5])))
= 1 : 4 : (filter ((== 1) ◦ ('mod' 3)) (map (^2) [4, 5]))
= 1 : 4 : (filter ((== 1) ◦ ('mod' 3)) (16 : (map (^2) [5])))
= 1 : 4 : 16 : (filter ((== 1) ◦ ('mod' 3)) (map (^2) [5]))
= 1 : 4 : 16 : (filter ((== 1) ◦ ('mod' 3)) (25 : (map (^2) [])))
= 1 : 4 : 16 : 25 : (filter ((== 1) ◦ ('mod' 3)) (map (^2) []))
```

## Problem: Zwischenergebnisse

Ein Beispiel:

```
filter ((== 1) ◦ ('mod' 3)) (map (^2) [1..5])
= filter ((== 1) ◦ ('mod' 3)) (1 : (map (^2) [2..5]))
= 1 : (filter ((== 1) ◦ ('mod' 3)) (map (^2) [2..5]))
= 1 : (filter ((== 1) ◦ ('mod' 3)) (4 : (map (^2) [3..5])))
= 1 : 4 : (filter ((== 1) ◦ ('mod' 3)) (map (^2) [3..5]))
= 1 : 4 : (filter ((== 1) ◦ ('mod' 3)) (9 : (map (^2) [4, 5])))
= 1 : 4 : (filter ((== 1) ◦ ('mod' 3)) (map (^2) [4, 5]))
= 1 : 4 : (filter ((== 1) ◦ ('mod' 3)) (16 : (map (^2) [5])))
= 1 : 4 : 16 : (filter ((== 1) ◦ ('mod' 3)) (map (^2) [5]))
= 1 : 4 : 16 : (filter ((== 1) ◦ ('mod' 3)) (25 : (map (^2) [])))
= 1 : 4 : 16 : 25 : (filter ((== 1) ◦ ('mod' 3)) (map (^2) []))
= 1 : 4 : 16 : 25 : (filter ((== 1) ◦ ('mod' 3)) [])
```

## Problem: Zwischenergebnisse

Ein Beispiel:

```
filter ((== 1) ◦ ('mod' 3)) (map (^2) [1..5])
= filter ((== 1) ◦ ('mod' 3)) (1 : (map (^2) [2..5]))
= 1 : (filter ((== 1) ◦ ('mod' 3)) (map (^2) [2..5]))
= 1 : (filter ((== 1) ◦ ('mod' 3)) (4 : (map (^2) [3..5])))
= 1 : 4 : (filter ((== 1) ◦ ('mod' 3)) (map (^2) [3..5]))
= 1 : 4 : (filter ((== 1) ◦ ('mod' 3)) (9 : (map (^2) [4, 5])))
= 1 : 4 : (filter ((== 1) ◦ ('mod' 3)) (map (^2) [4, 5]))
= 1 : 4 : (filter ((== 1) ◦ ('mod' 3)) (16 : (map (^2) [5])))
= 1 : 4 : 16 : (filter ((== 1) ◦ ('mod' 3)) (map (^2) [5]))
= 1 : 4 : 16 : (filter ((== 1) ◦ ('mod' 3)) (25 : (map (^2) [])))
= 1 : 4 : 16 : 25 : (filter ((== 1) ◦ ('mod' 3)) (map (^2) []))
= 1 : 4 : 16 : 25 : (filter ((== 1) ◦ ('mod' 3)) [])
= 1 : 4 : 16 : 25 : []
```

## Problem: Zwischenergebnisse

Ein Beispiel:

```
filter ((== 1) ◦ ('mod' 3)) (map (^2) [1..5])
= filter ((== 1) ◦ ('mod' 3)) (1 : (map (^2) [2..5]))
= 1 : (filter ((== 1) ◦ ('mod' 3)) (map (^2) [2..5]))
= 1 : (filter ((== 1) ◦ ('mod' 3)) (4 : (map (^2) [3..5])))
= 1 : 4 : (filter ((== 1) ◦ ('mod' 3)) (map (^2) [3..5]))
= 1 : 4 : (filter ((== 1) ◦ ('mod' 3)) (9 : (map (^2) [4, 5])))
= 1 : 4 : (filter ((== 1) ◦ ('mod' 3)) (map (^2) [4, 5]))
= 1 : 4 : (filter ((== 1) ◦ ('mod' 3)) (16 : (map (^2) [5])))
= 1 : 4 : 16 : (filter ((== 1) ◦ ('mod' 3)) (map (^2) [5]))
= 1 : 4 : 16 : (filter ((== 1) ◦ ('mod' 3)) (25 : (map (^2) [])))
= 1 : 4 : 16 : 25 : (filter ((== 1) ◦ ('mod' 3)) (map (^2) []))
= 1 : 4 : 16 : 25 : (filter ((== 1) ◦ ('mod' 3)) [])
= 1 : 4 : 16 : 25 : []
```

## Freies Theorem als Lösung?

Wir hatten, für jede Wahl von  $p$ ,  $h$  und  $l$ :

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

## Freies Theorem als Lösung?

Wir hatten, für jede Wahl von  $p$ ,  $h$  und  $l$ :

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

Warum nicht einfache linke Seite durch rechte Seite ersetzen?



## Freies Theorem als Lösung?

Wir hatten, für jede Wahl von  $p$ ,  $h$  und  $l$ :

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

Warum nicht einfache linke Seite durch rechte Seite ersetzen?

Probleme:

- immer noch zwei Durchläufe, bzw. Zwischenergebnis

## Freies Theorem als Lösung?

Wir hatten, für jede Wahl von  $p$ ,  $h$  und  $l$ :

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

Warum nicht einfache linke Seite durch rechte Seite ersetzen?

Probleme:

- immer noch zwei Durchläufe, bzw. Zwischenergebnis
- Gefahr von Doppelarbeit für  $h$

## Freies Theorem als Lösung?

Wir hatten, für jede Wahl von  $p$ ,  $h$  und  $l$ :

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

Warum nicht einfache linke Seite durch rechte Seite ersetzen?

Probleme:

- immer noch zwei Durchläufe, bzw. Zwischenergebnis
- Gefahr von Doppelarbeit für  $h$
- wenn Verlass nur auf Typ von `filter`:

## Freies Theorem als Lösung?

Wir hatten, für jede Wahl von  $p$ ,  $h$  und  $l$ :

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

Warum nicht einfache linke Seite durch rechte Seite ersetzen?

Probleme:

- immer noch zwei Durchläufe, bzw. Zwischenergebnis
- Gefahr von Doppelarbeit für  $h$
- wenn Verlass nur auf Typ von `filter`:
  - ▶ andere Funktion gleichen Typs könnte Liste **verlängern**

## Freies Theorem als Lösung?

Wir hatten, für jede Wahl von  $p$ ,  $h$  und  $l$ :

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

Warum nicht einfache linke Seite durch rechte Seite ersetzen?

Probleme:

- immer noch zwei Durchläufe, bzw. Zwischenergebnis
- Gefahr von Doppelarbeit für  $h$
- wenn Verlass nur auf Typ von `filter`:
  - ▶ andere Funktion gleichen Typs könnte Liste **verlängern**
  - ▶ ... und/oder **Elemente verdoppeln**

## Versuch Herleitung effizienter Lösung „per Hand“

Man ersetze zunächst einfach wie folgt:

$$\text{filter } p (\text{map } h l) \rightsquigarrow \text{filterMap } p h l$$

## Versuch Herleitung effizienter Lösung „per Hand“

Man ersetze zunächst einfach wie folgt:

$$\text{filter } p (\text{map } h l) \rightsquigarrow \text{filterMap } p h l$$

für eine angenommene neue Funktion:

$$\begin{aligned} \text{filterMap} &:: (b \rightarrow \text{Bool}) \rightarrow (a \rightarrow b) \rightarrow [a] \rightarrow [b] \\ \text{filterMap } p h l &= \text{filter } p (\text{map } h l) \end{aligned}$$

## Versuch Herleitung effizienter Lösung „per Hand“

Man ersetze zunächst einfach wie folgt:

$$\text{filter } p (\text{map } h l) \rightsquigarrow \text{filterMap } p h l$$

für eine angenommene neue Funktion:

$$\begin{aligned} \text{filterMap} &:: (b \rightarrow \text{Bool}) \rightarrow (a \rightarrow b) \rightarrow [a] \rightarrow [b] \\ \text{filterMap } p h l &= \text{filter } p (\text{map } h l) \end{aligned}$$

Dann, Verfeinerung und Optimierung dieser Funktion.



## Versuch Herleitung effizienter Lösung „per Hand“

Man ersetze zunächst einfach wie folgt:

$$\text{filter } p (\text{map } h l) \rightsquigarrow \text{filterMap } p h l$$

für eine angenommene neue Funktion:

$$\begin{aligned} \text{filterMap} &:: (b \rightarrow \text{Bool}) \rightarrow (a \rightarrow b) \rightarrow [a] \rightarrow [b] \\ \text{filterMap } p h l &= \text{filter } p (\text{map } h l) \end{aligned}$$

Dann, Verfeinerung und Optimierung dieser Funktion.

Zum Beispiel durch Instanziierung:

$$\begin{aligned} \text{filterMap} &:: (b \rightarrow \text{Bool}) \rightarrow (a \rightarrow b) \rightarrow [a] \rightarrow [b] \\ \text{filterMap } p h [] &= \text{filter } p (\text{map } h []) \\ \text{filterMap } p h (a : as) &= \text{filter } p (\text{map } h (a : as)) \end{aligned}$$

## Versuch Herleitung effizienter Lösung „per Hand“

Man ersetze zunächst einfach wie folgt:

$$\text{filter } p (\text{map } h l) \rightsquigarrow \text{filterMap } p h l$$

für eine angenommene neue Funktion:

$$\begin{aligned} \text{filterMap} &:: (b \rightarrow \text{Bool}) \rightarrow (a \rightarrow b) \rightarrow [a] \rightarrow [b] \\ \text{filterMap } p h l &= \text{filter } p (\text{map } h l) \end{aligned}$$

Dann, Verfeinerung und Optimierung dieser Funktion.

Zum Beispiel durch Instanziierung:

$$\begin{aligned} \text{filterMap} &:: (b \rightarrow \text{Bool}) \rightarrow (a \rightarrow b) \rightarrow [a] \rightarrow [b] \\ \text{filterMap } p h [] &= \text{filter } p (\text{map } h []) \\ \text{filterMap } p h (a : as) &= \text{filter } p (\text{map } h (a : as)) \end{aligned}$$

Und Anwendung bekannter Definitionen:

$$\begin{aligned} \text{filterMap} &:: (b \rightarrow \text{Bool}) \rightarrow (a \rightarrow b) \rightarrow [a] \rightarrow [b] \\ \text{filterMap } p h [] &= \text{filter } p [] \\ \text{filterMap } p h (a : as) &= \text{filter } p ((h a) : (\text{map } h as)) \end{aligned}$$

## Versuch Herleitung effizienter Lösung „per Hand“

Dann, Verfeinerung und Optimierung dieser Funktion.

Zum Beispiel durch Instanziierung:

```
filterMap :: (b → Bool) → (a → b) → [a] → [b]
filterMap p h []      = filter p (map h [])
filterMap p h (a : as) = filter p (map h (a : as))
```

Und Anwendung bekannter Definitionen:

```
filterMap :: (b → Bool) → (a → b) → [a] → [b]
filterMap p h []      = filter p []
filterMap p h (a : as) = filter p ((h a) : (map h as))
```

Gegebenenfalls wiederholt:

## Versuch Herleitung effizienter Lösung „per Hand“

Dann, Verfeinerung und Optimierung dieser Funktion.

Zum Beispiel durch Instanziierung:

```
filterMap :: (b → Bool) → (a → b) → [a] → [b]
filterMap p h []      = filter p (map h [])
filterMap p h (a : as) = filter p (map h (a : as))
```

Und Anwendung bekannter Definitionen:

```
filterMap :: (b → Bool) → (a → b) → [a] → [b]
filterMap p h []      = filter p []
filterMap p h (a : as) = filter p ((h a) : (map h as))
```

Gegebenenfalls wiederholt:

```
filterMap :: (b → Bool) → (a → b) → [a] → [b]
filterMap p h []      = []
filterMap p h (a : as) | p (h a)    = (h a) : (filter p (map h as))
                       | otherwise = filter p (map h as)
```

## Versuch Herleitung effizienter Lösung „per Hand“

Dann, Verfeinerung und Optimierung dieser Funktion.

Gegebenenfalls wiederholt:

```
filterMap :: (b → Bool) → (a → b) → [a] → [b]
```

```
filterMap p h [] = []
```

```
filterMap p h (a : as) | p (h a) = (h a) : (filter p (map h as))  
                       | otherwise = filter p (map h as)
```

## Versuch Herleitung effizienter Lösung „per Hand“

Dann, Verfeinerung und Optimierung dieser Funktion.

Gegebenenfalls wiederholt:

```
filterMap :: (b → Bool) → (a → b) → [a] → [b]
```

```
filterMap p h [] = []
```

```
filterMap p h (a : as) | p (h a) = (h a) : (filter p (map h as))  
                       | otherwise = filter p (map h as)
```

Wenn möglich, Rückgriff auf ursprünglich postulierte Gleichung:

```
filterMap :: (b → Bool) → (a → b) → [a] → [b]
```

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

## Versuch Herleitung effizienter Lösung „per Hand“

Dann, Verfeinerung und Optimierung dieser Funktion.

Gegebenenfalls wiederholt:

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

Wenn möglich, Rückgriff auf ursprünglich postulierte Gleichung:

```
filterMap :: (b -> Bool) -> (a -> b) -> [a] -> [b]
filterMap p h l = filter p (map h l)
```

Also:

```
filterMap :: (b -> Bool) -> (a -> b) -> [a] -> [b]
filterMap p h []      = []
filterMap p h (a : as) | p (h a)    = (h a) : (filterMap p h as)
                       | otherwise = filterMap p h as
```

## Probleme mit diesem Ansatz

Im Beispiel erzeugte Funktion:

```
filterMap :: (b → Bool) → (a → b) → [a] → [b]
```

```
filterMap p h [] = []
```

```
filterMap p h (a : as) | p (h a) = (h a) : (filterMap p h as)  
                       | otherwise = filterMap p h as
```

ist potentiell ineffizient wegen Verdopplung des Ausdrucks „(h a)“.



## Probleme mit diesem Ansatz

Im Beispiel erzeugte Funktion:

```
filterMap :: (b → Bool) → (a → b) → [a] → [b]
filterMap p h []      = []
filterMap p h (a : as) | p (h a)    = (h a) : (filterMap p h as)
                       | otherwise = filterMap p h as
```

ist potentiell ineffizient wegen Verdopplung des Ausdrucks „(h a)“.

Weitere Probleme im Allgemeinen:

- Termination der Transformation (und somit des Compilers)

## Probleme mit diesem Ansatz

Im Beispiel erzeugte Funktion:

```
filterMap :: (b → Bool) → (a → b) → [a] → [b]
filterMap p h []      = []
filterMap p h (a : as) | p (h a)    = (h a) : (filterMap p h as)
                       | otherwise = filterMap p h as
```

ist potentiell ineffizient wegen Verdopplung des Ausdrucks „(h a)“.

Weitere Probleme im Allgemeinen:

- Termination der Transformation (und somit des Compilers)
- Beliebige Berechnungen zur Compile-Zeit

## Probleme mit diesem Ansatz

Im Beispiel erzeugte Funktion:

```
filterMap :: (b → Bool) → (a → b) → [a] → [b]
filterMap p h []      = []
filterMap p h (a : as) | p (h a)    = (h a) : (filterMap p h as)
                       | otherwise = filterMap p h as
```

ist potentiell ineffizient wegen Verdopplung des Ausdrucks „(h a)“.

Weitere Probleme im Allgemeinen:

- Termination der Transformation (und somit des Compilers)
- Beliebige Berechnungen zur Compile-Zeit
- Geeigneter Rückgriff auf vorherige Definitionen nötig

## Probleme mit diesem Ansatz

Im Beispiel erzeugte Funktion:

```
filterMap :: (b → Bool) → (a → b) → [a] → [b]
filterMap p h []      = []
filterMap p h (a : as) | p (h a)    = (h a) : (filterMap p h as)
                       | otherwise = filterMap p h as
```

ist potentiell ineffizient wegen Verdopplung des Ausdrucks „(h a)“.

Weitere Probleme im Allgemeinen:

- Termination der Transformation (und somit des Compilers)
- Beliebige Berechnungen zur Compile-Zeit
- Geeigneter Rückgriff auf vorherige Definitionen nötig
- Wenn „unvorsichtig“, sogar Termination des Programms fraglich

## Probleme mit diesem Ansatz

Im Beispiel erzeugte Funktion:

```
filterMap :: (b → Bool) → (a → b) → [a] → [b]
filterMap p h []      = []
filterMap p h (a : as) | p (h a)    = (h a) : (filterMap p h as)
                       | otherwise = filterMap p h as
```

ist potentiell ineffizient wegen Verdopplung des Ausdrucks „(h a)“.

Weitere Probleme im Allgemeinen:

- Termination der Transformation (und somit des Compilers)
- Beliebige Berechnungen zur Compile-Zeit
- Geeigneter Rückgriff auf vorherige Definitionen nötig
- Wenn „unvorsichtig“, sogar Termination des Programms fraglich
- Einschränkungen nötig, sonst Aufwandsverdopplung möglich

## Probleme mit diesem Ansatz

Im Beispiel erzeugte Funktion:

```
filterMap :: (b → Bool) → (a → b) → [a] → [b]
filterMap p h []      = []
filterMap p h (a : as) | p (h a)    = (h a) : (filterMap p h as)
                       | otherwise = filterMap p h as
```

ist potentiell ineffizient wegen Verdopplung des Ausdrucks „(h a)“.

Weitere Probleme im Allgemeinen:

- Termination der Transformation (und somit des Compilers)
- Beliebige Berechnungen zur Compile-Zeit
- Geeigneter Rückgriff auf vorherige Definitionen nötig
- Wenn „unvorsichtig“, sogar Termination des Programms fraglich
- Einschränkungen nötig, sonst Aufwandsverdopplung möglich
- hoher Implementierungsaufwand

## Populäre Strategie: Deforestation [Wadler 1990]

Funktioniert zum Beispiel für `filter p (map h l)`, nach geeigneter Anpassung für higher-order.

## Populäre Strategie: Deforestation [Wadler 1990]

Funktioniert zum Beispiel für `filter p (map h l)`, nach geeigneter Anpassung für higher-order.

Aber bringt keine Verbesserung zum Beispiel für `map h (reverse l)` mit effizientem `reverse`:



## Populäre Strategie: Deforestation [Wadler 1990]

Funktioniert zum Beispiel für `filter p (map h l)`, nach geeigneter Anpassung für higher-order.

Aber bringt keine Verbesserung zum Beispiel für `map h (reverse l)` mit effizientem `reverse`:

```
mapReverse :: (a -> b) -> [a] -> [b]
mapReverse h l = map h (reverse l)
```

## Populäre Strategie: Deforestation [Wadler 1990]

Funktioniert zum Beispiel für `filter p (map h l)`, nach geeigneter Anpassung für higher-order.

Aber bringt keine Verbesserung zum Beispiel für `map h (reverse l)` mit effizientem `reverse`:

```
mapReverse :: (a -> b) -> [a] -> [b]
mapReverse h l = map h (reverse l)
mapReverse h l = map h (reverse' l [])
```

## Populäre Strategie: Deforestation [Wadler 1990]

Funktioniert zum Beispiel für `filter p (map h l)`, nach geeigneter Anpassung für higher-order.

Aber bringt keine Verbesserung zum Beispiel für `map h (reverse l)` mit effizientem `reverse`:

```
mapReverse :: (a -> b) -> [a] -> [b]
```

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

```
mapReverse h l = map h (reverse' l [])
```

```
mapReverse h l = mapReverse' h l []
```

```
mapReverse' :: (a -> b) -> [a] -> [a] -> [b]
```

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

## Populäre Strategie: Deforestation [Wadler 1990]

Funktioniert zum Beispiel für `filter p (map h l)`, nach geeigneter Anpassung für higher-order.

Aber bringt keine Verbesserung zum Beispiel für `map h (reverse l)` mit effizientem `reverse`:

```
mapReverse :: (a -> b) -> [a] -> [b]
```

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

```
mapReverse h l = map h (reverse' l [])
```

```
mapReverse h l = mapReverse' h l []
```

```
mapReverse' :: (a -> b) -> [a] -> [a] -> [b]
```

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

```
mapReverse' h [] l' = map h (reverse' [] l')
```

```
mapReverse' h (a : as) l' = map h (reverse' (a : as) l')
```

## Populäre Strategie: Deforestation [Wadler 1990]

Funktioniert zum Beispiel für `filter p (map h l)`, nach geeigneter Anpassung für higher-order.

Aber bringt keine Verbesserung zum Beispiel für `map h (reverse l)` mit effizientem `reverse`:

```
mapReverse :: (a -> b) -> [a] -> [b]
```

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

```
mapReverse h l = map h (reverse' l [])
```

```
mapReverse h l = mapReverse' h l []
```

```
mapReverse' :: (a -> b) -> [a] -> [a] -> [b]
```

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

```
mapReverse' h [] l' = map h (reverse' [] l')
```

```
mapReverse' h (a : as) l' = map h (reverse' (a : as) l')
```

```
mapReverse' h [] l' = map h l'
```

```
mapReverse' h (a : as) l' = map h (reverse' as (a : l'))
```

## Populäre Strategie: Deforestation [Wadler 1990]

Funktioniert zum Beispiel für `filter p (map h l)`, nach geeigneter Anpassung für higher-order.

Aber bringt keine Verbesserung zum Beispiel für `map h (reverse l)` mit effizientem `reverse`:

```
mapReverse :: (a -> b) -> [a] -> [b]
```

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

```
mapReverse h l = map h (reverse' l [])
```

```
mapReverse h l = mapReverse' h l []
```

```
mapReverse' :: (a -> b) -> [a] -> [a] -> [b]
```

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

```
mapReverse' h [] l' = map h (reverse' [] l')
```

```
mapReverse' h (a : as) l' = map h (reverse' (a : as) l')
```

```
mapReverse' h [] l' = map h l'
```

```
mapReverse' h (a : as) l' = map h (reverse' as (a : l'))
```

```
mapReverse' h (a : as) l' = mapReverse' h as (a : l')
```

## Populäre Strategie: Deforestation [Wadler 1990]

Funktioniert zum Beispiel für `filter p (map h l)`, nach geeigneter Anpassung für higher-order.

Aber bringt keine Verbesserung zum Beispiel für `map h (reverse l)` mit effizientem `reverse`:

`mapReverse :: (a -> b) -> [a] -> [b]`

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

`mapReverse h l = map h (reverse' l [])`

→ `mapReverse h l = mapReverse' h l []`

`mapReverse' :: (a -> b) -> [a] -> [a] -> [b]`

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

`mapReverse' h [] l' = map h (reverse' [] l')`

`mapReverse' h (a : as) l' = map h (reverse' (a : as) l')`

→ `mapReverse' h [] l' = map h l'`

`mapReverse' h (a : as) l' = map h (reverse' as (a : l'))`

→ `mapReverse' h (a : as) l' = mapReverse' h as (a : l')`

## Weniger Syntax — Mehr Struktur

Man könnte es ja auch mal mit `foldr`-fusion versuchen:

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

sofern für alle  $x$  und  $y$ ,

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



## Weniger Syntax — Mehr Struktur

Man könnte es ja auch mal mit `foldr`-fusion versuchen:

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

sofern für alle  $x$  und  $y$ ,

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

Also, umformulieren:

$$\text{map } h \ l = \text{foldr } (\lambda x \ ys \rightarrow (h \ x) : ys) \ [] \ l$$

## Weniger Syntax — Mehr Struktur

Man könnte es ja auch mal mit `foldr`-fusion versuchen:

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

sofern für alle  $x$  und  $y$ ,

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

Also, umformulieren:

$$\text{map } h \ l = \text{foldr } (\lambda x \ ys \rightarrow (h \ x) : ys) \ [] \ l$$

Folglich:

$$\text{filter } p \ (\text{map } h \ l) = \text{foldr } g \ (\text{filter } p \ []) \ l$$

sofern für alle  $x$  und  $ys$ ,

$$\text{filter } p \ ((h \ x) : ys) = g \ x \ (\text{filter } p \ ys)$$

## Weniger Syntax — Mehr Struktur

Die Bedingung

$$\text{filter } p ((h \ x) : ys) = g \ x (\text{filter } p \ ys)$$

lässt sich weiter analysieren:

## Weniger Syntax — Mehr Struktur

Die Bedingung

$$\text{filter } p ((h \ x) : ys) = g \ x (\text{filter } p \ ys)$$

lässt sich weiter analysieren:

$$\begin{aligned} & \text{filter } p ((h \ x) : ys) \\ = & \\ & g \ x (\text{filter } p \ ys) \end{aligned}$$

## Weniger Syntax — Mehr Struktur

Die Bedingung

$$\text{filter } p ((h \ x) : ys) = g \ x (\text{filter } p \ ys)$$

lässt sich weiter analysieren:

$$\begin{aligned} & \text{filter } p ((h \ x) : ys) \\ = & \text{let } y = h \ x \text{ in (if } p \ y \text{ then } y : (\text{filter } p \ ys) \text{ else } \text{filter } p \ ys) \\ & g \ x (\text{filter } p \ ys) \end{aligned}$$

## Weniger Syntax — Mehr Struktur

Die Bedingung

$$\text{filter } p ((h \ x) : ys) = g \ x (\text{filter } p \ ys)$$

lässt sich weiter analysieren:

$$\begin{aligned} & \text{filter } p ((h \ x) : ys) \\ = & \text{let } y = h \ x \text{ in (if } p \ y \text{ then } y : (\text{filter } p \ ys) \text{ else } \text{filter } p \ ys) \\ = & g \ x (\text{filter } p \ ys) \end{aligned}$$

bei Wahl von

$$g \ x \ ys' = \text{let } y = h \ x \text{ in (if } p \ y \text{ then } y : ys' \text{ else } ys')$$

## Weniger Syntax — Mehr Struktur

Die Bedingung

$$\text{filter } p ((h \ x) : ys) = g \ x (\text{filter } p \ ys)$$

lässt sich weiter analysieren:

$$\begin{aligned} & \text{filter } p ((h \ x) : ys) \\ = & \text{let } y = h \ x \text{ in (if } p \ y \text{ then } y : (\text{filter } p \ ys) \text{ else } \text{filter } p \ ys) \\ = & g \ x (\text{filter } p \ ys) \end{aligned}$$

bei Wahl von

$$g \ x \ ys' = \text{let } y = h \ x \text{ in (if } p \ y \text{ then } y : ys' \text{ else } ys')$$

Folglich, mit genau diesem  $g$ :

$$\text{filter } p (\text{map } h \ l) = \text{foldr } g \ [] \ l$$

## Weniger Syntax — Mehr Struktur

Die Bedingung

$$\text{filter } p ((h \ x) : ys) = g \ x (\text{filter } p \ ys)$$

lässt sich weiter analysieren:

$$\begin{aligned} & \text{filter } p ((h \ x) : ys) \\ = & \text{let } y = h \ x \text{ in (if } p \ y \text{ then } y : (\text{filter } p \ ys) \text{ else } \text{filter } p \ ys) \\ = & g \ x (\text{filter } p \ ys) \end{aligned}$$

bei Wahl von

$$g \ x \ ys' = \text{let } y = h \ x \text{ in (if } p \ y \text{ then } y : ys' \text{ else } ys')$$

Folglich, mit genau diesem  $g$ :

$$\text{filter } p (\text{map } h \ l) = \text{foldr } g \ [] \ l$$

Wenig überraschend (?) entspricht dieses `foldr g []` in etwa dem schon zuvor hergeleiteten `filterMap p h`.



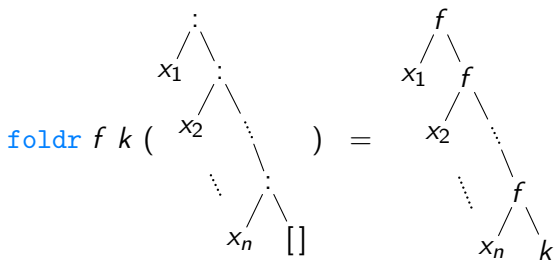
## Problem: Zwischenergebnisse

Ein Beispiel:

```
filter ((== 1) ◦ ('mod' 3)) (map (^2) [1..5])
= filter ((== 1) ◦ ('mod' 3)) (1 : (map (^2) [2..5]))
= 1 : (filter ((== 1) ◦ ('mod' 3)) (map (^2) [2..5]))
= 1 : (filter ((== 1) ◦ ('mod' 3)) (4 : (map (^2) [3..5])))
= 1 : 4 : (filter ((== 1) ◦ ('mod' 3)) (map (^2) [3..5]))
= 1 : 4 : (filter ((== 1) ◦ ('mod' 3)) (9 : (map (^2) [4, 5])))
= 1 : 4 : (filter ((== 1) ◦ ('mod' 3)) (map (^2) [4, 5]))
= 1 : 4 : (filter ((== 1) ◦ ('mod' 3)) (16 : (map (^2) [5])))
= 1 : 4 : 16 : (filter ((== 1) ◦ ('mod' 3)) (map (^2) [5]))
= 1 : 4 : 16 : (filter ((== 1) ◦ ('mod' 3)) (25 : (map (^2) [])))
= 1 : 4 : 16 : 25 : (filter ((== 1) ◦ ('mod' 3)) (map (^2) []))
= 1 : 4 : 16 : 25 : (filter ((== 1) ◦ ('mod' 3)) [])
= 1 : 4 : 16 : 25 : []
```

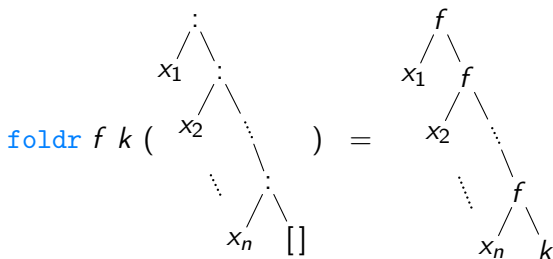
## Short Cut Deforestation [Gill et al. 1993]

Schreibe Listenkonsumenten mittels `foldr`:



## Short Cut Deforestation [Gill et al. 1993]

Schreibe Listenkonsumenten mittels `foldr`:

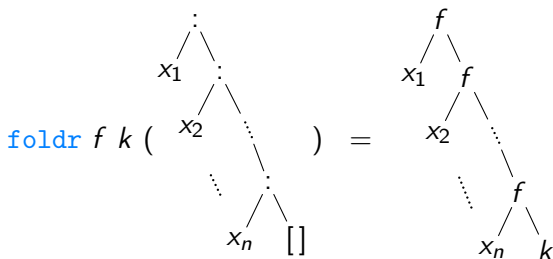


Schreibe Listenerzeuger mittels `build`:

```
build :: ( $\forall \beta. (\alpha \rightarrow \beta \rightarrow \beta) \rightarrow \beta \rightarrow \beta$ )  $\rightarrow \beta \rightarrow \beta$   $\rightarrow$  [ $\alpha$ ]  
build prod = prod (:) []
```

## Short Cut Deforestation [Gill et al. 1993]

Schreibe Listenkonsumenten mittels `foldr`:

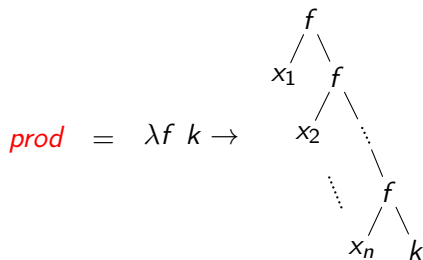


Schreibe Listenerzeuger mittels `build`:

`build` ::  $(\forall \beta. (\alpha \rightarrow \beta \rightarrow \beta) \rightarrow \beta \rightarrow \beta) \rightarrow [\alpha]$   
`build prod = prod (:) []`

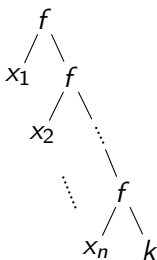
## Short Cut Deforestation [Gill et al. 1993]

Jedes derart polymorphe *prod* muss einer Funktion der folgenden Form entsprechen, für feste  $n \geq 0$  und  $x_1, \dots, x_n$ :



## Short Cut Deforestation [Gill et al. 1993]

Jedes derart polymorphe *prod* muss einer Funktion der folgenden Form entsprechen, für feste  $n \geq 0$  und  $x_1, \dots, x_n$ :

$$\textit{prod} = \lambda f k \rightarrow$$


Zum Beispiel:

```
filter :: (α → Bool) → [α] → [α]
filter p l = build (λf k → let f' x y | p x      = f x y
                                | otherwise = y
                                in foldr f' k l)
```

## Short Cut Deforestation [Gill et al. 1993]

Benutze folgende Regel:

$$\text{foldr } f \ k \ (\text{build } \textit{prod}) \rightsquigarrow \textit{prod } f \ k$$

## Short Cut Deforestation [Gill et al. 1993]

Benutze folgende Regel:

$$\text{foldr } f' \ k' \ (\text{build } \text{prod}) \rightsquigarrow \text{prod } f' \ k'$$



## Short Cut Deforestation [Gill et al. 1993]

Benutze folgende Regel:

$$\text{foldr } f' \ k' \ (\text{build } \textit{prod}) \rightsquigarrow \textit{prod } f' \ k'$$

Zur Rechtfertigung:

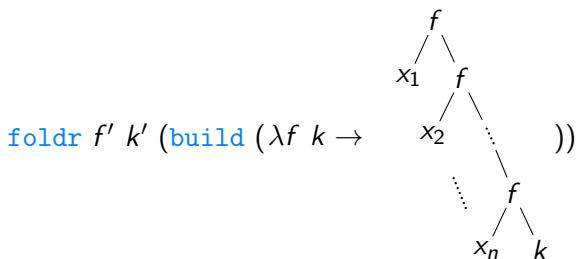
$$\text{foldr } f' \ k' \ (\text{build } \textit{prod})$$

## Short Cut Deforestation [Gill et al. 1993]

Benutze folgende Regel:

$$\text{foldr } f' \ k' \ (\text{build } \text{prod}) \rightsquigarrow \text{prod } f' \ k'$$

Zur Rechtfertigung:

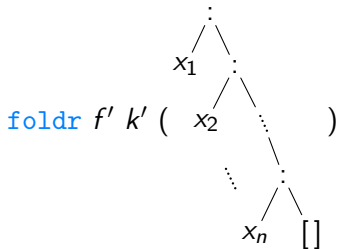


## Short Cut Deforestation [Gill et al. 1993]

Benutze folgende Regel:

$$\text{foldr } f' \ k' \ (\text{build } \text{prod}) \rightsquigarrow \text{prod } f' \ k'$$

Zur Rechtfertigung:

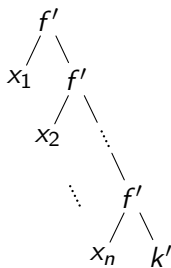


## Short Cut Deforestation [Gill et al. 1993]

Benutze folgende Regel:

$$\text{foldr } f' \ k' \ (\text{build } \text{prod}) \rightsquigarrow \text{prod } f' \ k'$$

Zur Rechtfertigung:

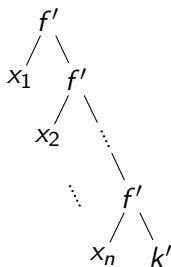


## Short Cut Deforestation [Gill et al. 1993]

Benutze folgende Regel:

$$\text{foldr } f' \ k' \ (\text{build } \text{prod}) \rightsquigarrow \text{prod } f' \ k'$$

Zur Rechtfertigung:



Für den Compiler (GHC):

```
{-# RULES "foldr/build"
  ∀(prod :: ∀β. (α → β → β) → β → β) f k.
  foldr f k (build prod) = prod f k    #-}
```

## Formaler Korrektheitsbeweis

Gegeben sei  $prod :: \forall\beta. (\tau \rightarrow \beta \rightarrow \beta) \rightarrow \beta \rightarrow \beta$ .

## Formaler Korrektheitsbeweis

Gegeben sei  $prod :: \forall\beta. (\tau \rightarrow \beta \rightarrow \beta) \rightarrow \beta \rightarrow \beta$ .

Dann:

$$(prod, prod) \in \forall\mathcal{R}. (id_{\tau} \rightarrow \mathcal{R} \rightarrow \mathcal{R}) \rightarrow (\mathcal{R} \rightarrow \mathcal{R})$$

## Formaler Korrektheitsbeweis

Gegeben sei  $prod :: \forall\beta. (\tau \rightarrow \beta \rightarrow \beta) \rightarrow \beta \rightarrow \beta$ .

Dann:

$$(prod, prod) \in \forall\mathcal{R}. (id_\tau \rightarrow \mathcal{R} \rightarrow \mathcal{R}) \rightarrow (\mathcal{R} \rightarrow \mathcal{R})$$

$$\Leftrightarrow \forall\mathcal{R}. \forall(f_1, f_2) \in (id_\tau \rightarrow \mathcal{R} \rightarrow \mathcal{R}). \forall(k_1, k_2) \in \mathcal{R}.$$

$$(prod_{\tau_1} f_1 k_1, prod_{\tau_2} f_2 k_2) \in \mathcal{R}$$

wegen Definition von  $\forall\mathcal{R}. \mathcal{F}(\mathcal{R})$  und  $\mathcal{R} \rightarrow \mathcal{S}$



## Formaler Korrektheitsbeweis

Gegeben sei  $prod :: \forall \beta. (\tau \rightarrow \beta \rightarrow \beta) \rightarrow \beta \rightarrow \beta$ .

Dann:

$$\begin{aligned} & (prod, prod) \in \forall \mathcal{R}. (id_{\tau} \rightarrow \mathcal{R} \rightarrow \mathcal{R}) \rightarrow (\mathcal{R} \rightarrow \mathcal{R}) \\ \Leftrightarrow & \forall \mathcal{R}. \forall (f_1, f_2) \in (id_{\tau} \rightarrow \mathcal{R} \rightarrow \mathcal{R}). \forall (k_1, k_2) \in \mathcal{R}. \\ & \qquad \qquad \qquad (prod_{\tau_1} f_1 k_1, prod_{\tau_2} f_2 k_2) \in \mathcal{R} \\ \Rightarrow & ((:), f_2) \in (id_{\tau} \rightarrow (foldr f_2 k_2) \rightarrow (foldr f_2 k_2)) \\ & \wedge ([], k_2) \in (foldr f_2 k_2) \\ \Rightarrow & (prod_{[\tau]} (:), prod_{\tau'} f_2 k_2) \in (foldr f_2 k_2) \\ & \text{durch Spezialisierung zu } \mathcal{R} = foldr f_2 k_2, f_1 = (:)\text{ und } k_1 = [] \end{aligned}$$

für jede mögliche Wahl von  $f_2 :: \tau \rightarrow \tau' \rightarrow \tau'$  und  $k_2 :: \tau'$ .

## Formaler Korrektheitsbeweis

Gegeben sei  $prod :: \forall \beta. (\tau \rightarrow \beta \rightarrow \beta) \rightarrow \beta \rightarrow \beta$ .

Dann:

$$\begin{aligned} & (prod, prod) \in \forall \mathcal{R}. (id_{\tau} \rightarrow \mathcal{R} \rightarrow \mathcal{R}) \rightarrow (\mathcal{R} \rightarrow \mathcal{R}) \\ \Leftrightarrow & \forall \mathcal{R}. \forall (f_1, f_2) \in (id_{\tau} \rightarrow \mathcal{R} \rightarrow \mathcal{R}). \forall (k_1, k_2) \in \mathcal{R}. \\ & \qquad \qquad \qquad (prod_{\tau_1} f_1 k_1, prod_{\tau_2} f_2 k_2) \in \mathcal{R} \\ \Rightarrow & ((:), f_2) \in (id_{\tau} \rightarrow (foldr f_2 k_2) \rightarrow (foldr f_2 k_2)) \\ & \wedge ([], k_2) \in (foldr f_2 k_2) \\ \Rightarrow & (prod_{[\tau]} (:) [], prod_{\tau'} f_2 k_2) \in (foldr f_2 k_2) \\ \Leftrightarrow & foldr f_2 k_2 (prod_{[\tau]} (:) []) = prod_{\tau'} f_2 k_2 \\ & \text{durch Vereinfachung} \end{aligned}$$

für jede mögliche Wahl von  $f_2 :: \tau \rightarrow \tau' \rightarrow \tau'$  und  $k_2 :: \tau'$ .

## Formaler Korrektheitsbeweis

Gegeben sei  $prod :: \forall \beta. (\tau \rightarrow \beta \rightarrow \beta) \rightarrow \beta \rightarrow \beta$ .

Dann:

$$\begin{aligned} & (prod, prod) \in \forall \mathcal{R}. (id_{\tau} \rightarrow \mathcal{R} \rightarrow \mathcal{R}) \rightarrow (\mathcal{R} \rightarrow \mathcal{R}) \\ \Leftrightarrow & \forall \mathcal{R}. \forall (f_1, f_2) \in (id_{\tau} \rightarrow \mathcal{R} \rightarrow \mathcal{R}). \forall (k_1, k_2) \in \mathcal{R}. \\ & \qquad \qquad \qquad (prod_{\tau_1} f_1 k_1, prod_{\tau_2} f_2 k_2) \in \mathcal{R} \\ \Rightarrow & ((:), f_2) \in (id_{\tau} \rightarrow (foldr f_2 k_2) \rightarrow (foldr f_2 k_2)) \\ & \wedge ([], k_2) \in (foldr f_2 k_2) \\ \Rightarrow & (prod_{[\tau]} (:) [], prod_{\tau'} f_2 k_2) \in (foldr f_2 k_2) \\ \Leftrightarrow & \mathbf{foldr} f_2 k_2 (prod_{[\tau]} (:) []) = prod_{\tau'} f_2 k_2 \\ \Leftrightarrow & \mathbf{foldr} f_2 k_2 (\mathbf{build} prod) = prod_{\tau'} f_2 k_2 \\ & \text{per Definition von } \mathbf{build} \end{aligned}$$

für jede mögliche Wahl von  $f_2 :: \tau \rightarrow \tau' \rightarrow \tau'$  und  $k_2 :: \tau'$ .

## Formaler Korrektheitsbeweis

Gegeben sei  $prod :: \forall \beta. (\tau \rightarrow \beta \rightarrow \beta) \rightarrow \beta \rightarrow \beta$ .

Dann:

$$\begin{aligned} & (prod, prod) \in \forall \mathcal{R}. (id_{\tau} \rightarrow \mathcal{R} \rightarrow \mathcal{R}) \rightarrow (\mathcal{R} \rightarrow \mathcal{R}) \\ \Leftrightarrow & \forall \mathcal{R}. \forall (f_1, f_2) \in (id_{\tau} \rightarrow \mathcal{R} \rightarrow \mathcal{R}). \forall (k_1, k_2) \in \mathcal{R}. \\ & \qquad \qquad \qquad (prod_{\tau_1} f_1 k_1, prod_{\tau_2} f_2 k_2) \in \mathcal{R} \\ \Rightarrow & ((:), f_2) \in (id_{\tau} \rightarrow (foldr f_2 k_2) \rightarrow (foldr f_2 k_2)) \\ & \wedge ([], k_2) \in (foldr f_2 k_2) \\ \Rightarrow & (prod_{[\tau]} (:), prod_{\tau'} f_2 k_2) \in (foldr f_2 k_2) \\ \Leftrightarrow & foldr f_2 k_2 (prod_{[\tau]} (:), []) = prod_{\tau'} f_2 k_2 \\ \Leftrightarrow & \mathbf{foldr} f_2 k_2 (\mathbf{build} prod) = prod_{\tau'} f_2 k_2 \end{aligned}$$

für jede mögliche Wahl von  $f_2 :: \tau \rightarrow \tau' \rightarrow \tau'$  und  $k_2 :: \tau'$ .

## Anwendung am Beispiel

Funktionen mittels `foldr` und `build` schreiben:

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

```
filter p l = build (λf k → let f' x y | p x      = f x y  
                                | otherwise = y  
                            in foldr f' k l)
```

```
map :: (α → β) → [α] → [β]
```

```
map h l = build (λf k → foldr (λx y → f (h x) y) k l)
```

## Anwendung am Beispiel

Funktionen mittels `foldr` und `build` schreiben:

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

```
filter p l = build (λf k → let f' x y | p x      = f x y  
                                | otherwise = y  
                                in foldr f' k l)
```

```
map :: (α → β) → [α] → [β]
```

```
map h l = build (λf k → foldr (λx y → f (h x) y) k l)
```

Dann:

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

## Anwendung am Beispiel

Funktionen mittels `foldr` und `build` schreiben:

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

```
filter p l = build (λf k → let f' x y | p x      = f x y  
                                | otherwise = y  
                                in foldr f' k l)
```

```
map :: (α → β) → [α] → [β]
```

```
map h l = build (λf k → foldr (λx y → f (h x) y) k l)
```

Dann:

```
filter p (map h l)  
= build (λf k →  
        let f' ... = ...  
        in foldr f' k  
        (build (λf k → foldr (λx y → f (h x) y) k l)))
```

## Anwendung am Beispiel

Funktionen mittels `foldr` und `build` schreiben:

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

```
filter p l = build (λf k → let f' x y | p x      = f x y  
                                | otherwise = y  
                                in foldr f' k l)
```

```
map :: (α → β) → [α] → [β]
```

```
map h l = build (λf k → foldr (λx y → f (h x) y) k l)
```

Dann:

```
    filter p (map h l)  
= build (λf k →  
        let f' ... = ...  
        in foldr f' k  
            (build (λf k → foldr (λx y → f (h x) y) k l)))  
= build (λf k →  
        let f' ... = ...  
        in foldr (λx y → f' (h x) y) k l)
```



## Bisher... (und Verweise auf Skript)

- Einführung und Haskell (Abschnitte 1, 2, 2.1 und halb 2.2)
- Gleichungsbasiertes Schließen und Induktion (Abschnitt 2.3)
- Freie Theoreme intuitiv (Abschnitt 3.1)
- Freie Theoreme formal (Abschnitt 8.2)
- Beweisprinzip für polymorphe Funktionen auf Listen
- Optimierung durch syntaktisches Umformen
- Short Cut Deforestation (Abschnitt 3.2)

## Hilft `foldr/build` denn immer?

Mal ein anderes Beispiel:

```
fromTo :: (Ord α, Enum α) ⇒ α → α → [α]
fromTo n m = go n
  where go i = if i > m then []
              else i : (go (succ i))

zip :: [α] → [β] → [(α, β)]
zip [] [] = []
zip (a : as) (b : bs) = (a, b) : (zip as bs)
```

## Hilft `foldr/build` denn immer?

Mal ein anderes Beispiel:

```
fromTo :: (Ord  $\alpha$ , Enum  $\alpha$ )  $\Rightarrow$   $\alpha \rightarrow \alpha \rightarrow [\alpha]$   
fromTo n m = go n  
  where go i = if i > m then []  
           else i : (go (succ i))
```

```
zip :: [ $\alpha$ ]  $\rightarrow$  [ $\beta$ ]  $\rightarrow$  [( $\alpha$ ,  $\beta$ )]  
zip [] [] = []  
zip (a : as) (b : bs) = (a, b) : (zip as bs)
```

Und dann ein Ausdruck mit **zwei** Zwischenergebnissen:

```
zip (fromTo 1 10) (fromTo 'a' 'j')
```

## Hilft `foldr/build` denn immer?

Mal ein anderes Beispiel:

```
fromTo :: (Ord α, Enum α) ⇒ α → α → [α]
fromTo n m = go n
  where go i = if i > m then []
              else i : (go (succ i))
```

```
zip :: [α] → [β] → [(α, β)]
zip [] [] = []
zip (a : as) (b : bs) = (a, b) : (zip as bs)
```

Und dann ein Ausdruck mit **zwei** Zwischenergebnissen:

```
zip (fromTo 1 10) (fromTo 'a' 'j')
```

Was nun?