Deskriptive Programmierung

Systematische Programmtransformation

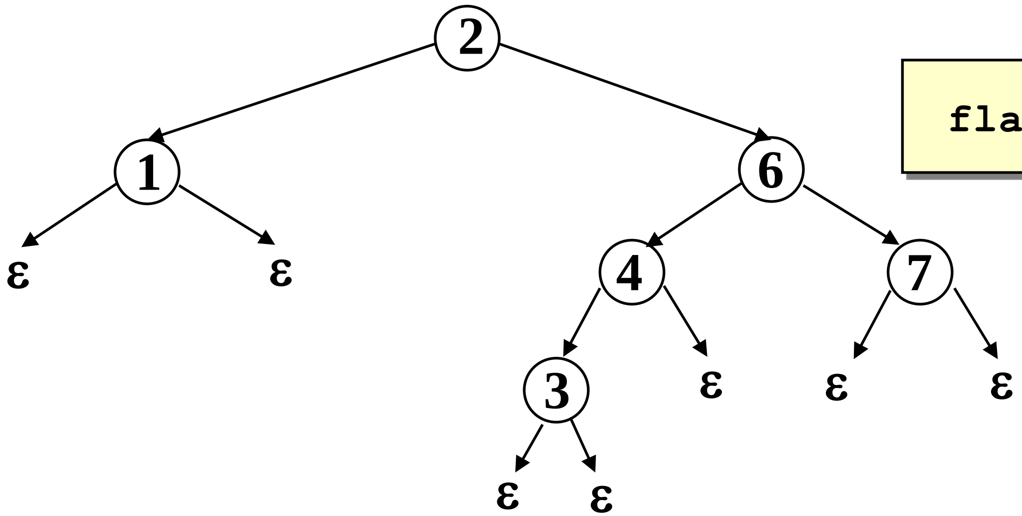
Ein der reverse-Problematik verwandtes Problem auf Binärbäumen

- Zur Erinnerung:

```
data BinTree a  = Empty | Node (BinTree a) a (BinTree a)
```

- Eine typische Traversalfunktion darauf:

```
flatten :: BinTree a → [a]
flatten Empty          = []
flatten (Node l a r) = flatten l ++ [a] ++ flatten r
```




```
flatten aTree = [1,2,3,4,6,7]
```

Das Problem sowohl bei reverse als auch bei flatten

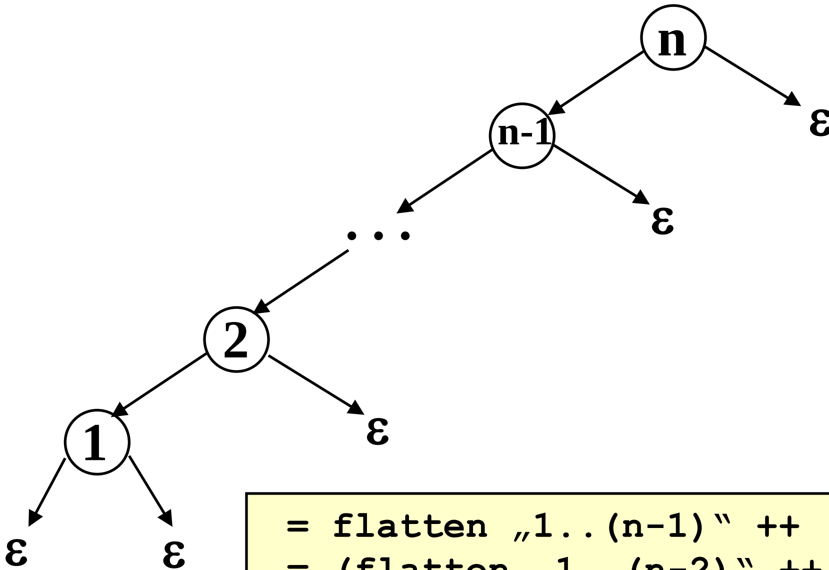
Die Effizienz der Durchläufe wird durch die Zahl der Reduktionen insbesondere bei der Konkatenation ++ bestimmt:

```
[1,2] ++ [3,4,5] = 1:([2] ++ [3,4,5])
                 = 1:(2:([] ++ [3,4,5]))
                 = 1:(2:[3,4,5])
```

Generell**:** Für die Auswertung von **xs ++ ys** mit **length xs = n** werden n + 1 Reduktionen via ++ benötigt!

Besonders problematische Fälle

„worst case" für **`flatten`**, links-entartete Bäume:



Zahl der Reduktionen:

$$\Sigma = \frac{n*(n+1)}{2} + 4n + 1$$

```
= flatten „1..(n-1)" ++ [n] ++ []
= (flatten „1..(n-2)" ++ [n-1] ++ []) ++ [n] ++ []
  . . .
= ((...(([] ++ [1]) ++ [2]) ++ ...) ++ [n-1]) ++ [n]
= ((...([1]          ++ [2]) ++ ...) ++ [n-1]) ++ [n]
= ((...[1,2]                  ++ ...) ++ [n-1]) ++ [n]
= ((...[1,2,3]                ++ ...) ++ [n-1]) ++ [n]
  . . .
= [1,2, . . ., n-1]                              ++ [n]
= [1,2, . . ., n]
```

4n+1

1 + 2 • • • + n

Ansatz zur Lösung

- Die Auswertung von **`flatten`** benötigt also $O(n^2)$ Schritte.
- Zur Verbesserung versuchen wir, die ++-Aufrufe zu eliminieren.
- Wir definieren dafür eine Funktion **`flattenCat`**, die einen Baum linearisiert und an das Ergebnis eine Liste anhängt:

$$\boxed{\texttt{flattenCat t as} =_{\texttt{def}} \texttt{flatten t ++ as}}$$

Spezifikation

- Für die Herleitung der Definition von **`flattenCat`** betrachten wir die zwei Fälle:
  - **`t = Empty`**
  - **`t = Node l a r`**

Herleitung von **`flattenCat`**

Fall **`t = Empty`**:

```
     flattenCat Empty as
  =  flatten Empty ++ as        [ Spezifikation ]
  =  [] ++ as                   [ Def. flatten 1. Fall ]
  =  as                         [ Def. (++) 1. Fall ]
```

Die erste zusammen mit der letzten Zeile liefern die erste definierende Gleichung für **`flattenCat`**:

```
flattenCat Empty as = as
flattenCat ...
```

Herleitung von **`flattenCat`**

Fall **`t = Node l a r`**:

```
    flattenCat (Node l a r) as
=   flatten (Node l a r) ++ as                     [ Spezifikation ]
=   (flatten l ++ [a] ++ flatten r) ++ as          [ Def. flatten 2. Fall ]
=   flatten l ++ [a] ++ flatten r ++ as            [ (++) assoziativ ]
=   flattenCat l ([a] ++ flatten r ++ as)          [ Spezifikation ]
=   flattenCat l ( a : flatten r ++ as )           [ Def. (++) ]
=   flattenCat l ( a : flattenCat r as )           [ Spezifikation ]
```

Finden einer effizienten Definition

- Damit ergibt sich die folgende Definition:

```
flattenCat :: BinTree a → [a] → [a]
flattenCat Empty        as = as
flattenCat (Node l a r) as = flattenCat l (a : flattenCat r as)
```

- Die ursprüngliche Funktion erhält man durch Spezialisierung von **`flattenCat`**:

```
flatten t = flattenCat t []
```

Laufzeit ???

wegen ursprünglicher Spezifikation:

$$\texttt{flattenCat t as} =_{\texttt{def}} \texttt{flatten t ++ as}$$

Wieder zurück zu einigen der vorigen Fragestellungen über Listen

- Wie angemerkt, kann der Compiler statt:

  ```
  concat (map f xs)
  ```

  eine Funktion concatMap benutzen, welche für f und xs statt obigem wie folgt:

  ```
  foldr ((++) . f) [ ] xs
  ```

  definiert ist.

- Die Korrektheit dessen kann man sicher per Induktion über xs beweisen (indem man explizite Definitionen von concat, map und foldr heranzieht).

- Schön wäre, wenn man stattdessen die Definitionen von concat und map benutzt, um eine passende für concatMap automatisch herzuleiten.

- Oder wenn man concat und/oder map mittels foldr ausdrückt, und dann das foldr ((++) . f) [ ] xs automatisch erhält.

Explizite syntaktische Herleitung einer neuen Definition

- Zunächst der „explizite Weg“. Mittels:

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

```
concat  [ ]           =   [ ]
concat  (xs : xss)    =   xs ++ concat xss
```

leite eine neue Funktion her, so dass:

```
concatMap f xs  =  concat (map f xs)
```

- Notwendig sind offenbar zwei Fälle:

```
concatMap f  [ ]        =  ???
concatMap f  (x : xs)   =  ???
```

Explizite syntaktische Herleitung einer neuen Definition

- Postuliert:

```
concatMap f xs  =  concat (map f xs)
```

- Herleitung:

```
concatMap f [ ]       =  concat (map f [ ])
                      =  concat [ ]
                      =  [ ]

concatMap f (x : xs)  =  concat (map f (x : xs))
                      =  concat (f x : map f xs)
                      =  f x ++ concat (map f xs)
                      =  f x ++ concatMap f xs
```

Alternativ, Verwendung von foldr

Oder aber:

- map und concat mittels foldr ausdrückt:

$$\boxed{\text{map f} = \text{foldr } (\backslash x\ ys \rightarrow f\ x : ys)\ [\ ]}$$

$$\boxed{\text{concat} = \text{foldr } (++)\ [\ ]}$$

- Nun Versuch, daraus eine foldr-basierte Definition zu erhalten für:

$$\boxed{\text{concatMap f} = \text{concat . map f}}$$

- Offenkundig relevante Frage: Wann ist eine (komponierte) Funktion überhaupt mittels foldr ausdrückbar?

Die „Universaleigenschaft“ von foldr

- Wir hatten dass, wann immer es möglich ist, eine Funktion in folgende Form zu bringen:

```
g [ ]       = k
g (x : xs)  = f  x (g xs)
```

für irgendwelche k und f

dann gilt: `g  =  foldr  f  k`

- Noch stärker kann man formulieren, dass für jede Funktion g gilt:

$$\boxed{g = \text{foldr } f \; k} \quad \Leftrightarrow \quad \boxed{g\ [\,] = k \;\wedge\; \forall x,xs.\ g\ (x : xs) = f\ x\ (g\ xs)}$$

Herleitung einer allgemeinen Fusionsregel

- Wegen

$$\boxed{g \;=\; \text{foldr}\;\; f\;\; k} \quad \Leftrightarrow \quad \boxed{g\;[\,]\;=\;k \;\;\wedge\;\; \forall x,xs.\; g\,(x : xs) \;=\; f\;\; x\,(g\;xs)}$$

gilt:

$$\boxed{\text{foldr}\; f_2\; k_2 \;.\; \text{foldr}\; f_1\; k_1 \;=\; \text{foldr}\; f_3\; k_3}$$

$$\Leftrightarrow \quad \boxed{\begin{aligned}&(\text{foldr}\; f_2\; k_2 \;.\; \text{foldr}\; f_1\; k_1)\;\;[\,] \;=\; k_3 \;\;\wedge\;\; \forall x,xs.\; (\text{foldr}\; f_2\; k_2 \;.\; \text{foldr}\; f_1\; k_1)\;(x : xs)\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad = \; f_3\; x\; ((\text{foldr}\; f_2\; k_2 \;.\; \text{foldr}\; f_1\; k_1)\; xs)\end{aligned}}$$

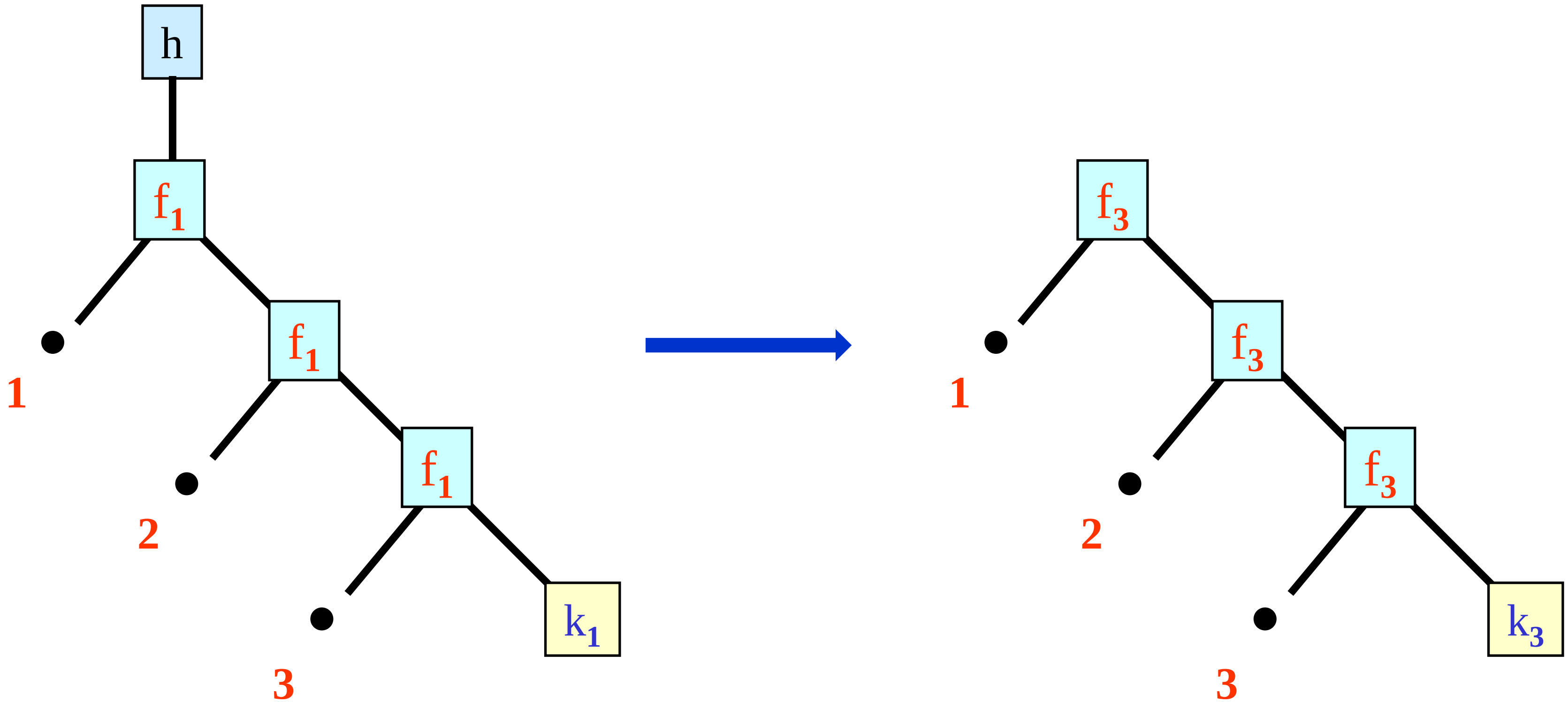
$$\Leftrightarrow \quad \boxed{\begin{aligned}&\text{foldr}\; f_2\; k_2\; k_1 \;= k_3 \;\;\wedge\;\; \forall x,xs.\; \text{foldr}\; f_2\; k_2\; (f_1\; x\; (\text{foldr}\; f_1\; k_1\; xs))\\ &\qquad\qquad\qquad\qquad\qquad = f_3\; x\; (\text{foldr}\; f_2\; k_2\; (\text{foldr}\; f_1\; k_1\; xs))\end{aligned}}$$

$$\Leftarrow \quad \boxed{\text{foldr}\; f_2\; k_2\; k_1 \;= k_3 \;\;\wedge\;\; \forall x,y.\; \text{foldr}\; f_2\; k_2\; (f_1\; x\; y) \;= f_3\; x\; (\text{foldr}\; f_2\; k_2\; y)}$$

- Bzw., allgemeiner:

$$\boxed{h \;.\; \text{foldr}\; f_1\; k_1 \;=\; \text{foldr}\; f_3\; k_3} \quad \Leftarrow \quad \boxed{h\;\; k_1 \;=\; k_3 \;\;\wedge\;\; \forall x,y.\; h\; (f_1\; x\; y) \;=\; f_3\; x\; (h\; y)}$$

Visualisierung der allgemeinen Fusionsregel



$$h \,.\, \mathrm{foldr}\ f_1\ k_1 \;=\; \mathrm{foldr}\ f_3\ k_3 \quad \Leftarrow \quad h\ \ k_1 \;=\; k_3 \;\;\wedge\;\; \forall x,y.\ h\ (f_1\ x\ y) \;=\; f_3\ x\ (h\ y)$$

Wieder am konkreten Beispiel

- Anwendung der Fusionsregel

$$\boxed{h\ .\ \mathrm{foldr}\ f_1\ k_1\ =\ \mathrm{foldr}\ f_3\ k_3} \quad \Leftarrow \quad \boxed{h\ k_1\ =\ k_3\ \wedge\ \forall x,y.\ h\ (f_1\ x\ y)\ =\ f_3\ x\ (h\ y)}$$

auf:

$$\boxed{\mathrm{concatMap}\ f\ =\ \mathrm{concat}\ .\ \mathrm{map}\ f}$$

wobei:

$$\boxed{\mathrm{map}\ f\ =\ \mathrm{foldr}\ (\backslash x\ y \rightarrow f\ x : y)\ [\,]}$$

- Instanziierung zur Anwendung der Regel: $h\ =\ \mathrm{concat}$
  $f_1\ =\ \backslash x\ y \rightarrow f\ x : y$
  $k_1\ =\ [\,]$

- Folglich, geeignete Wahl: $k_3\ =\ [\,]$
  $f_3\ =\ \backslash x\ y' \rightarrow f\ x\ {+}{+}\ y'$

- Also:

$$\boxed{\mathrm{concatMap}\ f\ =\ \mathrm{foldr}\ (\backslash x\ y' \rightarrow f\ x\ {+}{+}\ y')\ [\,]}$$

Exkurs: Verallgemeinerung von foldr

- Wenn foldr denn so nützlich ist, wäre es vielleicht gut, ähnliche Funktionalität für andere Typen zu haben, zum Beispiel für:

```
data BinTree a  = Empty | Node (BinTree a) a (BinTree a)
```

- Nichts leichter als das:

```
tfold :: (b → a → b → b) → b → BinTree a → b

tfold f k Empty          = k
tfold f k (Node l a r) = f (tfold f k l) a (tfold f k r)
```

- Dann zum Beispiel:

```
size, depth :: BinTree a → Int
size   = tfold (\l _ r → 1 + l + r)  0
depth  = tfold (\l _ r → 1 + max l r)  0

flatten :: BinTree a → [a]
flatten = tfold (\l a r → l ++ [a] ++ r)  []
```