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:

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

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:

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

für eine angenommene neue Funktion:

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

Dann, Verfeinerung und Optimierung dieser Funktion.

Zum Beispiel durch Instanziierung:

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

Und Anwendung bekannter Definitionen:

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

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 „$(h\ a)$“.

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.

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

$$
\begin{aligned}
& \texttt{mapReverse} :: (a \to b) \to [a] \to [b] \\
& \texttt{mapReverse}\ h\ l = \texttt{map}\ h\ (\texttt{reverse}\ l) \\
& \texttt{mapReverse}\ h\ l = \texttt{map}\ h\ (\texttt{reverse}'\ l\ [\,]) \\
\to\ & \texttt{mapReverse}\ h\ l = \texttt{mapReverse}'\ h\ l\ [\,]
\end{aligned}
$$

$$
\begin{aligned}
& \texttt{mapReverse}' :: (a \to b) \to [a] \to [a] \to [b] \\
& \texttt{mapReverse}'\ h\ l\ l' = \texttt{map}\ h\ (\texttt{reverse}'\ l\ l') \\
& \texttt{mapReverse}'\ h\ [\,]\ \ \ \ \ \ l' = \texttt{map}\ h\ (\texttt{reverse}'\ [\,]\ l') \\
& \texttt{mapReverse}'\ h\ (a : as)\ l' = \texttt{map}\ h\ (\texttt{reverse}'\ (a : as)\ l') \\
\to\ & \texttt{mapReverse}'\ h\ [\,]\ \ \ \ \ \ l' = \texttt{map}\ h\ l' \\
& \texttt{mapReverse}'\ h\ (a : as)\ l' = \texttt{map}\ h\ (\texttt{reverse}'\ as\ (a : l')) \\
\to\ & \texttt{mapReverse}'\ h\ (a : as)\ l' = \texttt{mapReverse}'\ h\ as\ (a : l')
\end{aligned}
$$

Weniger Syntax — Mehr Struktur

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

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

sofern für alle x und y,

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

Also, umformulieren:

$$\texttt{map}\ h\ l = \texttt{foldr}\ (\lambda x\ ys \to (h\ x) : ys)\ [\,]\ l$$

Folglich:

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

sofern für alle x und ys,

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

Weniger Syntax — Mehr Struktur

Die Bedingung

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

lässt sich weiter analysieren:

$$\begin{array}{ll} & \texttt{filter}\ p\ ((h\ x) : ys) \\ = & \textbf{let}\ y = h\ x\ \textbf{in}\ (\textbf{if}\ p\ y\ \textbf{then}\ y : (\texttt{filter}\ p\ ys)\ \textbf{else}\ \texttt{filter}\ p\ ys) \\ = & g\ x\ (\texttt{filter}\ p\ ys) \end{array}$$

bei Wahl von

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

Folglich, mit genau diesem g:

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

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

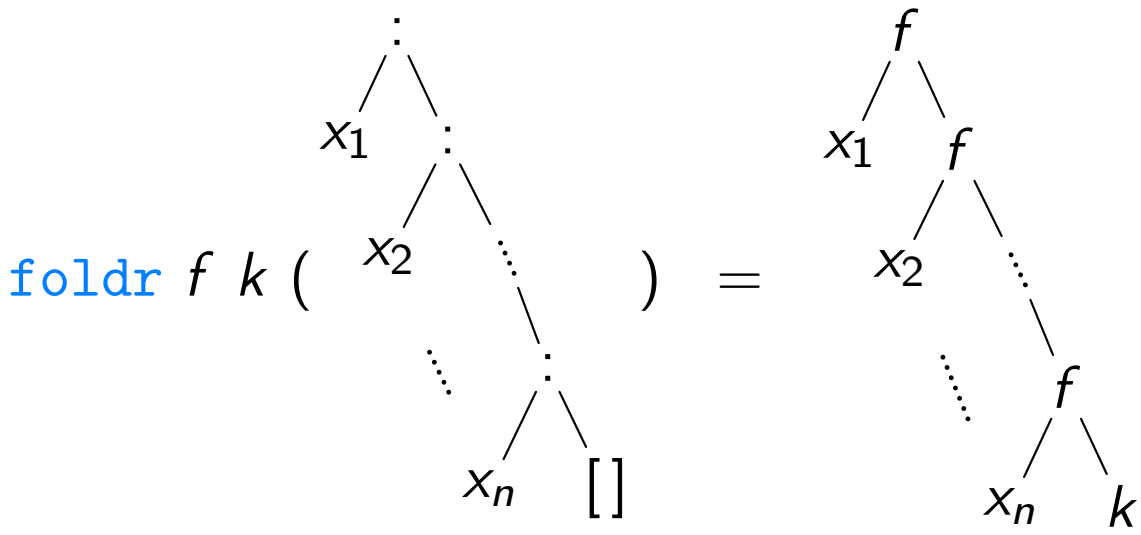
Problem: Zwischenergebnisse

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`:



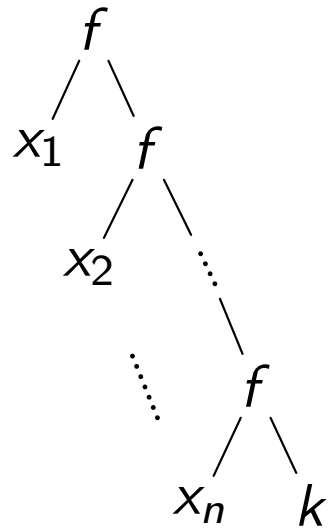
Schreibe Listenerzeuger mittels `build`:

$$\texttt{build} :: (\forall\beta.\ (\alpha \to \beta \to \beta) \to \beta \to \beta) \to [\alpha]$$
$$\texttt{build}\ \mathit{prod} = \mathit{prod}\ (:)\ []$$

Short Cut Deforestation [Gill et al. 1993]

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

$$prod \quad = \quad \lambda f\ k \rightarrow$$



Zum Beispiel:

$$\begin{aligned}
&\texttt{filter} :: (\alpha \rightarrow \text{Bool}) \rightarrow [\alpha] \rightarrow [\alpha] \\
&\texttt{filter}\ p\ l = \texttt{build}\ (\lambda f\ k \rightarrow \textbf{let}\ f'\ x\ y \mid p\ x \qquad\qquad = f\ x\ y \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid \texttt{otherwise} = y \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{in}\ \texttt{foldr}\ f'\ k\ l)
\end{aligned}$$

Short Cut Deforestation [Gill et al. 1993]

Benutze folgende Regel:

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

Zur Rechtfertigung:

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

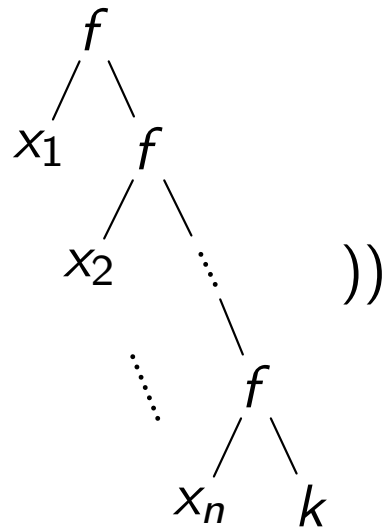
Short Cut Deforestation [Gill et al. 1993]

Benutze folgende Regel:

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

Zur Rechtfertigung:

$$\texttt{foldr}\ f'\ k'\ (\texttt{build}\ (\lambda f\ k \rightarrow$$



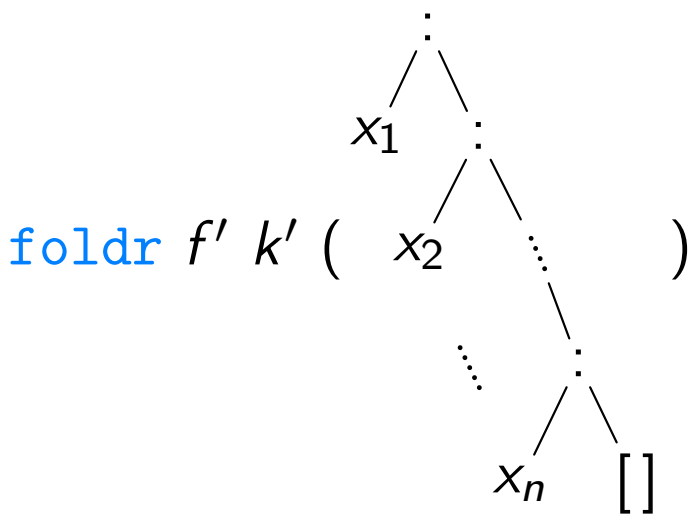
$$))$$

Short Cut Deforestation [Gill et al. 1993]

Benutze folgende Regel:

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

Zur Rechtfertigung:

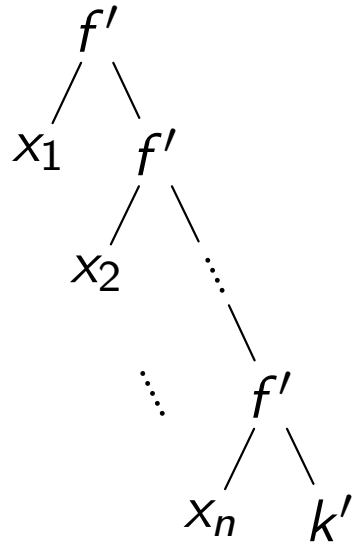


Short Cut Deforestation [Gill et al. 1993]

Benutze folgende Regel:

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

Zur Rechtfertigung:



Für den Compiler (GHC):

$$\{-\#\ \textbf{RULES}\ \text{“foldr/build”}$$
$$\forall(prod :: \forall\beta.\ (\alpha \to \beta \to \beta) \to \beta \to \beta)\ f\ k.$$
$$\texttt{foldr}\ f\ k\ (\texttt{build}\ prod) = prod\ f\ k \qquad \#-\}$$

Formaler Korrektheitsbeweis

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

Dann:

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

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

Anwendung am Beispiel

Funktionen mittels `foldr` und `build` schreiben:

$$
\begin{aligned}
&\texttt{filter} :: (\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha] \\
&\texttt{filter}\ p\ l = \texttt{build}\ (\lambda f\ k \to \textbf{let}\ f'\ x\ y \mid p\ x \qquad\quad = f\ x\ y \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid \texttt{otherwise} = y \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{in}\ \texttt{foldr}\ f'\ k\ l) \\
&\texttt{map} :: (\alpha \to \beta) \to [\alpha] \to [\beta] \\
&\texttt{map}\ h\ l = \texttt{build}\ (\lambda f\ k \to \texttt{foldr}\ (\lambda x\ y \to f\ (h\ x)\ y)\ k\ l)
\end{aligned}
$$

Dann:

$$
\begin{aligned}
& \texttt{filter}\ p\ (\texttt{map}\ h\ l) \\
= \quad & \texttt{build}\ (\lambda f\ k \to \\
& \qquad \textbf{let}\ f'\ \cdots = \cdots \\
& \qquad \textbf{in}\ \texttt{foldr}\ f'\ k \\
& \qquad\quad (\texttt{build}\ (\lambda f\ k \to \texttt{foldr}\ (\lambda x\ y \to f\ (h\ x)\ y)\ k\ l))) \\
= \quad & \texttt{build}\ (\lambda f\ k \to \\
& \qquad \textbf{let}\ f'\ \cdots = \cdots \\
& \qquad \textbf{in}\ \texttt{foldr}\ (\lambda x\ y \to f'\ (h\ x)\ y)\ k\ l)
\end{aligned}
$$

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)
```

Und dann ein Ausdruck mit zwei Zwischenergebnissen:

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

Was nun?