4.5 Beseitigung von Zwischendatenstrukturen

- Funktionale Programmierer lieben es, Zwischenergebnisse in Listen zu sammeln, die mit höheren Funktionen weiter verarbeitet werden.
- Solche höheren Funktionen sind etwa:

$$
\begin{array}{lllllllll}
\mathsf{map} & = & \textbf{fn}\ f \ \Rightarrow & \mathsf{fix}\,(m, \textbf{fn}\ l \ \Rightarrow \ \textbf{case}\ l\ \textbf{of}\ [\,]\ :\ [\,] \\
 & & & \quad \mid \ \ ::\ z\ :\ \textbf{case}\ z\ \textbf{of}\ (x, xs)\ : & \textbf{let} & z_1 & = & f\ x \\
 & & & & & z_2 & = & m\ xs \\
 & & & & & z' & = & (z_1, z_2) \\
 & & & & \textbf{in} & ::\ z')
\end{array}
$$

$$
\begin{array}{lllll}
\mathsf{filter} & = & \mathbf{fn}\ p \ \Rightarrow & \mathsf{fix}\,(f, \mathbf{fn}\ l \ \Rightarrow \ \mathbf{case}\ l\ \mathbf{of}\ [\,] \ : \ [\,] \\
& & & \mid \ ::z \ : \ \ \mathbf{case}\ z\ \mathbf{of}\ (x, xs) \ : \\
& & & \qquad\qquad \mathbf{if}\ p\,x\ \mathbf{then} \ \ \mathbf{let} \ \ z_2 \ = \ f\ xs \\
& & & \qquad\qquad\qquad\qquad\qquad\qquad\quad z' \ = \ (x, z_2) \\
& & & \qquad\qquad\qquad\qquad\qquad \mathbf{in}\ ::z' \\
& & & \qquad\qquad \mathbf{else}\ f\ xs \\
\mathsf{foldl} & = & \mathbf{fn}\ f \ \Rightarrow & \mathsf{fix}\,(h, \mathbf{fn}\ a \ \Rightarrow \ \mathbf{fn}\ l \ \Rightarrow \ \mathbf{case}\ l\ \mathbf{of}\ [\,] \ : \ a \\
& & & \mid ::z \ : \ \ \mathbf{case}\ z\ \mathbf{of}\ (x, xs) \ : \ \ \mathbf{let} \ \ a' \ = \ f\ a\ x \\
& & & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{in}\ h\ a'\ xs)
\end{array}
$$

$$
\begin{aligned}
\textsf{id} &= \textbf{fn}\ x \Rightarrow x \\
\textsf{comp} &= \textbf{fn}\ f \Rightarrow \textbf{fn}\ g \Rightarrow \textbf{fn}\ x \Rightarrow \begin{array}[t]{l} \textbf{let}\ y = g\ x \\ \textbf{in}\ f\ y \end{array} \\
\textsf{comp}_1 &= \textbf{fn}\ f \Rightarrow \textbf{fn}\ g \Rightarrow \textbf{fn}\ x_1 \Rightarrow \textbf{fn}\ x_2 \Rightarrow \\
&\qquad \textbf{let}\ y = g\ x_1 \\
&\qquad \textbf{in}\ f\ y\ x_2 \\
\textsf{comp}_2 &= \textbf{fn}\ f \Rightarrow \textbf{fn}\ g \Rightarrow \textbf{fn}\ x_1 \Rightarrow \textbf{fn}\ x_2 \Rightarrow \\
&\qquad \textbf{let}\ y = g\ x_2 \\
&\qquad \textbf{in}\ f\ x_1\ y
\end{aligned}
$$

Beispiel:

$$
\begin{array}{lcl}
\mathsf{sum} & = & \mathsf{foldl}\ (+)\ 0 \\
\mathsf{length} & = & \mathbf{let}\ f\ =\ \mathsf{map}\ (\mathbf{fn}\ x \to 1) \\
 & & \mathbf{in}\ \ \mathsf{comp}\ \mathsf{sum}\ f \\
\mathsf{dev} & = & \mathbf{fn}\ l\ \Rightarrow\ \ \mathbf{let}\ \begin{array}[t]{lcl} s_1 & = & \mathsf{sum}\ l \\ n & = & \mathsf{length}\ l \\ mean & = & s_1/n \\ l_1 & = & \mathsf{map}\ (\mathbf{fn}\ x \Rightarrow x - mean)\ l \\ l_2 & = & \mathsf{map}\ (\mathbf{fn}\ x \Rightarrow x \cdot x)\ l_1 \\ s_2 & = & \mathsf{sum}\ l_2 \end{array} \\
 & & \qquad\qquad \mathbf{in}\ \ s_2/n
\end{array}
$$

Beobachtungen:

- Um das Programm auf eine Seite zu kriegen, wurden nicht alle Funktionsanwendungen in paarweise Anwendungen zerlegt.
- Die Formulierung ist gewöhnungsbedürftig :-)
- Explizite Rekursion taucht in dem Beispiel gar nicht mehr auf!
- Die Implementierung legt unnötige Zwischendatenstrukturen an!

  length könnte auch so implementiert werden:

$$\begin{array}{lll} \mathsf{length} & = & \mathbf{let}\ f\ =\ \mathbf{fn}\ a \Rightarrow \mathbf{fn}\ x \Rightarrow a+1 \\ & & \mathbf{in}\ \ \mathsf{foldl}\ f\ 0 \end{array}$$

- Diese Implementierung vermeidet eine Liste als Zwischendatenstruktur !!!

Vereinfachungsregeln:

$$
\begin{array}{lcl}
\mathsf{comp}\ \mathsf{id}\ f & = & \mathsf{comp}\ f\ \mathsf{id} \quad = \quad f \\
\mathsf{comp}_1\ f\ \mathsf{id} & = & \mathsf{comp}_2\ f\ \mathsf{id} \quad = \quad f \\
\mathsf{map}\ \mathsf{id} & = & \mathsf{id} \\
\mathsf{comp}\ (\mathsf{map}\ f)\ (\mathsf{map}\ g) & = & \mathsf{map}\ (\mathsf{comp}\ f\ g) \\
\mathsf{comp}\ (\mathsf{foldl}\ f\ a)\ (\mathsf{map}\ g) & = & \mathsf{foldl}\ (\mathsf{comp}_2\ f\ g)\ a
\end{array}
$$

Vereinfachungsregeln:

$$
\begin{array}{lcl}
\mathsf{comp}\ \mathsf{id}\ f & = & \mathsf{comp}\ f\ \mathsf{id} \quad = \quad f \\
\mathsf{comp}_1\ f\ \mathsf{id} & = & \mathsf{comp}_2\ f\ \mathsf{id} \quad = \quad f \\
\mathsf{map}\ \mathsf{id} & = & \mathsf{id} \\
\mathsf{comp}\ (\mathsf{map}\ f)\ (\mathsf{map}\ g) & = & \mathsf{map}\ (\mathsf{comp}\ f\ g) \\
\mathsf{comp}\ (\mathsf{foldl}\ f\ a)\ (\mathsf{map}\ g) & = & \mathsf{foldl}\ (\mathsf{comp}_2\ f\ g)\ a \\
\mathsf{comp}\ (\mathsf{filter}\ p_1)\ (\mathsf{filter}\ p_2) & = & \mathsf{filter}\ (\mathbf{fn}\ x \Rightarrow\ \ \mathbf{if}\ p_2\ x\ \mathbf{then}\ p_1\ x \\
 & & \qquad\qquad\qquad\quad \mathbf{else}\ \mathsf{false}) \\
\mathsf{comp}\ (\mathsf{foldl}\ f\ a)\ (\mathsf{filter}\ p) & = & \mathbf{let}\ h = \mathbf{fn}\ a \Rightarrow \mathbf{fn}\ x \Rightarrow\ \ \mathbf{if}\ p\ x\ \mathbf{then}\ f\ a\ x \\
 & & \qquad\qquad\qquad\qquad\qquad\quad\ \mathbf{else}\ a \\
 & & \mathbf{in}\ \ \mathsf{foldl}\ h\ a
\end{array}
$$

Achtung:

Anstelle von Funktionskompositionen können auch geschachtelte Funktionsanwendungen vorkommen ...

$$
\begin{array}{lcl}
\mathsf{id}\; x & = & x \\
\mathsf{map}\;\mathsf{id}\; l & = & l \\
\mathsf{map}\; f\; (\mathsf{map}\; g\; l) & = & \mathsf{map}\; (\mathsf{comp}\; f\; g)\; l \\
\mathsf{foldl}\; f\; a\; (\mathsf{map}\; g\; l) & = & \mathsf{foldl}\; (\mathsf{comp}_2\; f\; g)\; a\; l \\
\mathsf{filter}\; p_1\; (\mathsf{filter}\; p_2\; l) & = & \mathsf{filter}\; (\mathbf{fn}\; x \Rightarrow p_1\, x \wedge p_2\, x)\; l \\
\mathsf{foldl}\; f\; a\; (\mathsf{filter}\; p\; l) & = & \mathbf{let}\; h = \mathbf{fn}\; a \Rightarrow \mathbf{fn}\; x \Rightarrow \;\; \mathbf{if}\; p\, x\; \mathbf{then}\; f\, a\, x \\
 & & \qquad\qquad\qquad\qquad\qquad\quad \mathbf{else}\; a \\
 & & \mathbf{in}\;\; \mathsf{foldl}\; h\; a\; l
\end{array}
$$

Beispiel, optimiert:

$$
\begin{array}{lcl}
\mathsf{sum} & = & \mathsf{foldl}\ (+)\ 0 \\
\mathsf{length} & = & \mathbf{let}\ \ f\ \ =\ \ \mathsf{comp_2}\ (+)\ (\mathbf{fn}\ x \to 1) \\
 & & \mathbf{in}\ \ \mathsf{foldl}\ f\ 0 \\
\mathsf{dev} & = & \mathbf{fn}\ l\ \Rightarrow\ \ \mathbf{let}\ \begin{array}[t]{lcl}
s_1 & = & \mathsf{sum}\ l \\
n & = & \mathsf{length}\ l \\
mean & = & s_1/n \\
f & = & \mathsf{comp}\ \ (\mathbf{fn}\ x \Rightarrow x \cdot x) \\
 & & \qquad\quad\ (\mathbf{fn}\ x \Rightarrow x - mean) \\
g & = & \mathsf{comp_2}\ (+)\ f \\
s_2 & = & \mathsf{foldl}\ g\ 0\ l
\end{array} \\
 & & \qquad\qquad\ \mathbf{in}\ \ s_2/n
\end{array}
$$

Bemerkungen:

- Sämtliche Zwischenlisten sind verschwunden :-)
- Es bleiben nur foldl – d.h. Schleifen :-))
- Funktionskompositionen können nun im nächsten Schritt durch Inlining weiter vereinfacht werden.
- Dann ergibt sich etwa innerhalb dev:

$$
\begin{array}{rclllrcl}
g & = & \textbf{fn}\ a & \Rightarrow & \textbf{fn}\ x \Rightarrow & \textbf{let}\ \ x_1 & = & x - \mathit{mean} \\
 & & & & & x_2 & = & x_1 \cdot x_1 \\
 & & & & & \multicolumn{3}{l}{\textbf{in}\ \ a + x_2}
\end{array}
$$

- Das Ergebnis ist eine Folge von **let**-Definitionen !!!

Erweiterung: Tabellierung

Wird die Liste durch Tabellierung einer Funktion hergestellt, kann deren Aufbau unter Umständen ganz vermieden werden ...

$$
\begin{array}{lllll}
\mathsf{tabulate} & = & \mathbf{fn}\ f\ \Rightarrow & \mathbf{fn}\ n\ \Rightarrow & \\
 & & & \mathbf{let}\ h = & \mathbf{fix}\,(t, \mathbf{fn}\ j\ \Rightarrow \\
 & & & & \quad \mathbf{if}\ j \geq n\ \mathbf{then}\ [] \\
 & & & & \quad \mathbf{else}\ \ \mathbf{let}\ \ x \ \ = \ \ f\ j \\
 & & & & \qquad\qquad\quad xs \ = \ \ t\ (j+1) \\
 & & & & \qquad\qquad\quad z \ \ \ = \ \ (x, xs) \\
 & & & & \qquad\quad \mathbf{in}\ \ ::z) \\
 & & & \mathbf{in}\ \ h\ 0 &
\end{array}
$$

Dann gilt:

$$\begin{array}{lcl} \mathsf{comp}\ (\mathsf{map}\ f)\ (\mathsf{tabulate}\ g) & = & \mathsf{tabulate}\ (\mathsf{comp}\ f\ g) \\ \mathsf{comp}\ (\mathsf{foldl}\ f\ a)\ (\mathsf{tabulate}\ g) & = & \mathsf{loop}\ (\mathsf{comp}_2\ f\ g)\ a \end{array}$$

Dabei ist:

$$\begin{array}{lcl} \mathsf{loop} & = & \mathbf{fn}\ f\ \Rightarrow\ \ \mathbf{fn}\ a\ \Rightarrow\ \mathbf{fn}\ n\ \Rightarrow \\ & & \quad \mathbf{let}\ h\ =\ \mathsf{fix}\,(t,\ \ \mathbf{fn}\ j \Rightarrow \mathbf{fn}\ a \Rightarrow \\ & & \qquad\qquad\qquad \mathbf{if}\ j \geq n\ \mathbf{then}\ a \\ & & \qquad\qquad\qquad \mathbf{else}\ t\ (j+1)\ (f\ a\ j)) \\ & & \quad \mathbf{in}\ \ h\ 0\ a \end{array}$$

Erweiterung (2): List-Reverse

Gelegentlich wird die Reihenfolge in einer Liste umgedreht:

$$
\begin{array}{lcl}
\mathsf{rev} & = & \textbf{let } r \; = \; \mathsf{fix}\,(h,\; \textbf{fn}\, a \Rightarrow \textbf{fn}\, l \Rightarrow \\
& & \qquad\qquad \textbf{case } l \textbf{ of } [\,] \;:\; a \\
& & \qquad\qquad \mid \; :: z \;:\; \textbf{case } z \textbf{ of } (x, xs) \;: \\
& & \qquad\qquad\qquad\qquad \textbf{let } a' = ::(x, a) \\
& & \qquad\qquad\qquad\qquad \textbf{in } \; h\; a'\; xs) \\
& & \textbf{in } \; r\,[\,]
\end{array}
$$

$$
\mathsf{foldr}\; f\; a \;=\; \mathsf{comp}\;(\mathsf{foldl}\; f\; a)\; \mathsf{rev}
$$

Diskussion:

- Die Standard-Implementierung von foldr ist nicht end-rekursiv.
- Die letzte Gleichung zerlegt ein foldr in zwei end-rekursive Funktionen — zu dem Preis, dass eine Zwischenliste angelegt wird.
- Vermutlich ist darum die Standard-Implementierung schneller :-)
- Die Operation rev kann jedoch möglicherweise weg-optimiert werden ...

Es gilt:

$$
\begin{array}{lcl}
\text{comp rev rev} & = & \text{id} \\
\text{comp rev } (\text{map } f) & = & \text{comp } (\text{map } f) \text{ rev} \\
\text{comp rev } (\text{filter } p) & = & \text{comp } (\text{filter } p) \text{ rev} \\
\text{comp rev } (\text{tabulate } f) & = & \text{rev\_tabulate } f
\end{array}
$$

Dabei tabelliert rev_tabulate in umgedrehter Reihenfolge. Diese Funktion erfüllt ganz ähnliche Eigenschaften wie tabulate:

$$
\begin{array}{lcl}
\text{comp } (\text{map } f\ a)\ (\text{rev\_tabulate } g) & = & \text{rev\_tabulate } (\text{comp}_2\ f\ g)\ a \\
\text{comp } (\text{foldl } f\ a)\ (\text{rev\_tabulate } g) & = & \text{rev\_loop } (\text{comp}_2\ f\ g)\ a
\end{array}
$$

Erweiterung (3): Index-Abhängigkeiten

- Die Korrektheit zeigt man mit Induktion über die Längen der auftretenden Listen.
- Ähnliche Kompositionsresultate kann man ausnutzen für Transformationen, die den Index mit einbeziehen:

$$
\begin{array}{lcll}
\mathsf{mapi} & = & \mathbf{fn}\ f\ \Rightarrow & \mathbf{let}\ h = \mathsf{fix}\,(m, \\
& & & \quad \mathbf{fn}\ i\ \Rightarrow\ \mathbf{fn}\ l\ \Rightarrow\ \mathbf{case}\ l\ \mathbf{of}\ [\,]\ :\ [\,] \\
& & & \quad \mid\ ::\ z\ :\ \mathbf{case}\ z\ \mathbf{of}\ (x, xs)\ :\ \ \mathbf{let}\ \begin{array}[t]{lcl} z_1 & = & f\ i\ x \\ i' & = & i+1 \\ z_2 & = & m\ i'\ xs \\ z' & = & (z_1, z_2) \end{array} \\
& & & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{in}\ ::\ z') \\
& & & \mathbf{in}\ \ h\ 0
\end{array}
$$

Analog gibt es index-abhängige Akkumulation:

$$
\begin{array}{lllll}
\mathsf{foldli} & = & \mathbf{fn}\ g \Rightarrow & \multicolumn{2}{l}{\mathbf{let}\ f = \mathsf{fix}\,(h, \mathbf{fn}\ i \Rightarrow \mathbf{fn}\ a \Rightarrow \mathbf{fn}\ l \Rightarrow} \\
 & & & \multicolumn{2}{l}{\mathbf{case}\ l\ \mathbf{of}\ [\,]\ :\ a} \\
 & & & \mid\ ::z\ :\ \ \mathbf{case}\ z\ \mathbf{of}\ (x, xs)\ : & \mathbf{let}\ \ a' = g\ i\ a\ x \\
 & & & & \qquad\ \ i' = i + 1 \\
 & & & & \mathbf{in}\ h\ i'\ a'\ xs) \\
 & & & \mathbf{in}\ \ f\ 0\ a &
\end{array}
$$

Bei der Komposition muss beachtet werden, dass jeweils die gleichen Indizes verwendet werden. Das erledigt: