Script generated by TTT

Title: Seidl: Programmoptimierung (23.01.2013)

Date: Wed Jan 23 08:30:52 CET 2013

Duration: 91:59 min

Pages: 51

Idea 1:

→ First, we introduce unique variable names.

→ Then, we only substitute functions which are staticly within the scope of the same global variables as the application :-)

→ For every expression, we determine all function definitions with this property :-)

$$(\text{fun } \underline{x} \to e)\ (\underline{e_2}) \overset{\beta}{\Longrightarrow} e[x := \underline{e_2}]$$

$$f = \text{fun } x \to 1 + x$$

$$f\ (\underline{g\ x}) \overset{\beta}{\Longrightarrow} 1 + g(x)$$

$$(\underline{\text{fun } x \to e})\ e_2 \overset{\beta}{\Longrightarrow} \text{let } \underline{x} = \underline{e_2} \text{ in } \underline{e}$$ ML

$$(\text{fun } \underline{x} \to e)\ (\underline{e_2}) \overset{\beta}{\Longrightarrow} e[x := \underline{e_2}]$$

$$f = \text{fun } x \to 1 + x$$

$$f\ (\underline{g\ x}) \overset{\beta}{\Longrightarrow} 1 + g(x)$$

$$(\underline{\text{fun } x \to e})\ e_2 \overset{\beta}{\Longrightarrow} \text{let } \underline{x} = \underline{e_2} \text{ in } \underline{e}$$ ML

In all other cases, $\mathcal{D}$ is propagated to the sub-expressions unchanged :-)

$(\text{fun } x \to e)\ (e_2) \overset{\beta}{\Longrightarrow} e[x := e_2]$

$f = \text{fun } x \to 1 + x$

... in the Example:

$f\ (g\ x) \overset{\beta}{\Longrightarrow} 1 + g(x)$

$$
\begin{aligned}
&\textbf{let} && x = 1\\
&\textbf{in let} && f = \textbf{let}\ \ x_1 = 2\\
& && \qquad \textbf{in}\ \ \textbf{fun}\ y \to y + x_1\\
&\textbf{in} && f\ x
\end{aligned}
$$

$(\text{fun } x \to e)\ e_2 \Longrightarrow \text{let } x = e_2 \text{ in } e$

... the application $f\ x$ is not in the scope of x_1

$\Longrightarrow$ we first duplicate the definition of x_1 :

$\text{let } f = \text{let } g = \ldots \text{ in fun } x \to e$

$$
\begin{aligned}
&\textbf{let} && x = 1\\
&\textbf{in let} && x_1 = 2\\
&\textbf{in let} && f = \textbf{let}\ \ x_1 = 2\\
& && \qquad \textbf{in}\ \ \textbf{fun}\ y \to y + x_1\\
&\textbf{in} && f\ x
\end{aligned}
$$

$\text{in } f\ x + f\ y$

$\Longrightarrow$ the inner definition becomes redundant !!!

$$
\begin{aligned}
&\textbf{let} && x = 1\\
&\textbf{in let} && x_1 = 2\\
&\textbf{in let} && f = \textbf{fun}\ y \to y + x_1\\
&\textbf{in} && f\ x
\end{aligned}
$$

$\Longrightarrow$ now we can apply inlining :

$$
\begin{aligned}
&\textbf{let} && x = 1\\
&\textbf{in let} && x_1 = 2\\
&\textbf{in let} && f = \textbf{fun}\ y \to y + x_1\\
&\textbf{in} && \boxed{\begin{aligned}&\textbf{let}\ \ y = x\\ &\textbf{in}\ \ \ y + x_1\end{aligned}}
\end{aligned}
$$

Removing variable-variable-assignments, we arrive at:

$$
\begin{array}{ll}
\text{let} & x = 1 \\
\text{in let} & x_1 = 2 \\
\text{in let} & f = \text{fun } y \rightarrow y + x_1 \\
\text{in} & \boxed{\begin{array}{ll} \text{let} & y = x \\ \text{in} & y + x_1 \end{array}}
\end{array}
$$

Removing variable-variable-assignments, we arrive at:

$$
\begin{array}{ll}
\text{let} & x = 1 \\
\text{in let} & x_1 = 2 \\
\text{in let} & f = \text{fun } y \rightarrow y + x_1 \\
\text{in} & \boxed{x + x_1}
\end{array}
$$

Idea 2:

→ We apply our value analysis.

→ We ignore global variables :-)

→ We only substitute functions without free variables :-))

Example: The map-Function

$$
\begin{array}{lll}
\text{let rec} & \mathsf{f} = \text{fun } x \rightarrow x \cdot x & \\
\text{and} & \mathsf{map} = \text{fun } g \rightarrow \text{fun } x \rightarrow \text{match } x & \\
& \quad \text{with } [\,] \rightarrow [\,] & \\
& \quad \mid \ x{::}xs \rightarrow g\ x :: \mathsf{map}\ g\ xs & \\
\text{in} & \mathsf{map}\ \mathsf{f}\ \mathit{list} &
\end{array}
$$

- The actual parameter f in the application map g is always fun $x \rightarrow x \cdot x$:-)
- Therefore, map g can be specialized to a new function h defined by:

$$
\begin{array}{lll}
\mathsf{h} & = & \text{let } g = \boxed{\text{fun } x \rightarrow x \cdot x} \\
& & \text{in fun } x \rightarrow \text{match } x \\
& & \quad \text{with } [\,] \rightarrow [\,] \\
& & \quad \mid \ x{::}xs \rightarrow g\ x :: \boxed{\mathsf{map}\ g}\ xs
\end{array}
$$

- The actual parameter f in the application map g is always fun $x \to x \cdot x$:-)
- Therefore, map g can be specialized to a new function h defined by:

fun g → let map_f = map f

map_f

$$
\begin{array}{lll}
\mathsf{h} = & \textbf{let } g = \boxed{\textbf{fun } x \to x \cdot x} & \\
& \textbf{in fun } x \to \textbf{match } x & \\
& \quad \textbf{with } [\,] \to [\,] & \\
& \quad \mid x::xs \to g\ x :: \boxed{\mathsf{map}\ g}\ xs &
\end{array}
$$

h g

map_f =

The inner occurrence of map g can be replaced with h

$\Longrightarrow$ fold-Transformation :-)

$$
\begin{array}{lll}
\mathsf{h} = & \textbf{let } g = \textbf{fun } x \to x \cdot x & \\
& \textbf{in fun } x \to \textbf{match } x & \\
& \quad \textbf{with } [\,] \to [\,] & \\
& \quad \mid x::xs \to g\ x :: \mathsf{h}\ xs &
\end{array}
$$

Inlining the function g yields:

$$
\begin{array}{lll}
\mathsf{h} = & \textbf{let } g = \textbf{fun } x \to x \cdot x & \\
& \textbf{in fun } x \to \textbf{match } x & \\
& \quad \textbf{with } [\,] \to [\,] & \\
& \quad \mid x::xs \to (\textbf{let } x = x & \\
& \qquad\qquad \textbf{in } x * x) :: \mathsf{h}\ xs &
\end{array}
$$

Removing useless definitions and variable-variable assignments yields:

$$
\begin{array}{lll}
\mathsf{h} = & \textbf{fun } x \to \textbf{match } x & \\
& \quad \textbf{with } [\,] \to [\,] & \\
& \quad \mid x::xs \to x * x :: \mathsf{h}\ xs &
\end{array}
$$

4.5 Deforestation

- Functional programmers love to collect intermediate results in lists which are processed by higher-order functions.
- Examples of such higher-order functions are:

$$\begin{array}{rcl} \textsf{map} & = & \textbf{fun}\ f \to \textbf{fun}\ l \to \textbf{match}\ l\ \textbf{with}\ [\,] \to [\,] \\ & & \qquad \mid x{::}xs \to f\ x :: \textsf{map}\ f\ xs) \end{array}$$

- The actual parameter f in the application map g is always **fun** $x \to x \cdot x$:-)
- Therefore, map g can be specialized to a new function h defined by:

$$\begin{array}{rcl} \textsf{h} & = & \textbf{let}\ g = \boxed{\textbf{fun}\ x \to x \cdot x} \\ & & \textbf{in}\ \ \textbf{fun}\ x \to \textbf{match}\ x \\ & & \quad \textbf{with}\ [\,] \to \quad [\,] \\ & & \quad \mid x{::}xs \to g\ x :: \boxed{\textsf{map}\ g}\ xs \end{array}$$

4.5 Deforestation

- Functional programmers love to collect intermediate results in lists which are processed by higher-order functions.
- Examples of such higher-order functions are:

map f [x_1; x_2; ...] = [f x_1; f x_2; ...]

$$\begin{array}{rcl} \textsf{map} & = & \textbf{fun}\ f \to \textbf{fun}\ l \to \textbf{match}\ l\ \textbf{with}\ [\,] \to [\,] \\ & & \qquad \mid x{::}xs \to f\ x :: \textsf{map}\ f\ xs) \end{array}$$

filter : (α → bool) → α list → α list

$$\begin{array}{rcl} \textsf{filter} & = & \textbf{fun}\ p \to \textbf{fun}\ l \to \textbf{match}\ l\ \textbf{with}\ [\,] \to [\,] \\ & & \qquad \mid x{::}xs \to \textbf{if}\ p\ x\ \textbf{then}\ x :: \textsf{filter}\ p\ xs \\ & & \qquad\qquad \textbf{else}\ \textsf{filter}\ p\ xs) \end{array}$$

filter p l = [x | x ∈ l, p x]

$$\begin{array}{rcl} \textsf{foldl} & = & \textbf{fun}\ f \to \textbf{fun}\ a \to \textbf{fun}\ l \to \textbf{match}\ l\ \textbf{with}\ [\,] \to a \\ & & \qquad \mid x{::}xs \to \textsf{foldl}\ f\ (f\ a\ x)\ xs) \end{array}$$

foldl ⊕ a [x_1; x_2; ...] = ((a ⊕ x_1) ⊕ x_2)...

$$\begin{array}{lcl} \textsf{filter} & = & \textbf{fun } p \;\to\; \textbf{fun } l \;\to\; \textbf{match } l \textbf{ with } [\,] \;\to\; [\,] \\ & & \quad\quad\quad\quad \mid x{::}xs \;\to\; \textbf{if } p\,x \textbf{ then } x :: \textsf{filter}\; p\; xs \\ & & \quad\quad\quad\quad\quad\quad\quad\quad \textbf{else } \textsf{filter}\; p\; xs) \end{array}$$

$$\begin{array}{lcl} \textsf{foldl} & = & \textbf{fun } f \;\to\; \textbf{fun } a \;\to\; \textbf{fun } l \;\to\; \textbf{match } l \textbf{ with } [\,] \;\to\; a \\ & & \quad\quad\quad\quad\quad\quad \mid x{::}xs \;\to\; \textsf{foldl}\; f\; (f\; a\; x)\; xs) \end{array}$$

l n a

$$\begin{array}{lcl} \textsf{filter} & = & \textbf{fun } p \;\to\; \textbf{fun } l \;\to\; \textbf{match } l \textbf{ with } [\,] \;\to\; [\,] \\ & & \quad\quad\quad\quad \mid x{::}xs \;\to\; \textbf{if } p\,x \textbf{ then } x :: \textsf{filter}\; p\; xs \\ & & \quad\quad\quad\quad\quad\quad\quad\quad \textbf{else } \textsf{filter}\; p\; xs) \end{array}$$

$$\begin{array}{lcl} \textsf{foldl} & = & \textbf{fun } f \;\to\; \textbf{fun } a \;\to\; \textbf{fun } l \;\to\; \textbf{match } l \textbf{ with } [\,] \;\to\; a \\ & & \quad\quad\quad\quad\quad\quad \mid x{::}xs \;\to\; \textsf{foldl}\; f\; (f\; a\; x)\; xs) \end{array}$$

$\underline{f}\ x\ \underline{(\text{foldr } f\ xs)}$

$g + f$

$g \cdot f$

$g \circ f$

$$\begin{array}{lcl} \underline{\textsf{id}} & = & \underline{\textbf{fun } x \;\to\; x} \\[1ex] \underline{\textsf{comp}} & = & \underline{\textbf{fun } f \;\to\; \textbf{fun } g \;\to\; \textbf{fun } x \;\to\; f\;(g\;x)} \\[1ex] \underline{\textsf{comp}_1} & = & \textbf{fun } f \;\to\; \textbf{fun } g \;\to\; \textbf{fun } x_1 \;\to\; \textbf{fun } x_2 \;\to \\ & & \quad f\;\underline{(g\;x_1)}\;\underline{x_2} \\[1ex] \textsf{comp}_2 & = & \textbf{fun } f \;\to\; \textbf{fun } g \;\to\; \textbf{fun } x_1 \;\to\; \textbf{fun } x_2 \;\to \\ & & \quad f\;x_1\;\underline{(g\;x_2)} \end{array}$$

$[x_1, x_2, x_3] \leadsto [1, 1, 1]$

Example:

$$\begin{array}{lcl} \textsf{sum} & = & \textsf{foldl}\;(+)\;0 \\ \textsf{length} & = & \textbf{let } \;f \;=\; \textsf{map}\;(\textbf{fun } x \to 1) \\ & & \textbf{in } \;\textsf{comp sum}\; f \\ \textsf{dev} & = & \textbf{fun } l \;\to\; \textbf{let } \;\begin{array}{lcl} s_1 & = & \textsf{sum}\; l \\ n & = & \textsf{length}\; l \\ mean & = & s_1/n \\ l_1 & = & \textsf{map}\;(\textbf{fun } x \to x - mean)\; l \\ l_2 & = & \textsf{map}\;(\textbf{fun } x \to x \cdot x)\; l_1 \\ s_2 & = & \textsf{sum}\; l_2 \end{array} \\ & & \quad\quad\quad\; \textbf{in } \;s_2/n \end{array}$$

fun x → sum(f x)

Example:

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

Observations:

- Explicit recursion does no longer occur!
- The implementation creates unnecessary intermediate data-structures!

  length could also be implemented as:

$$
\begin{aligned}
\mathsf{length} &= \textbf{let}\ f = \textbf{fun}\ a \to \textbf{fun}\ x \to a+1\\
&\quad\ \textbf{in}\ \mathsf{foldl}\ f\ 0
\end{aligned}
$$

- This implementation avoids to create intermediate lists !!!

Observations:

- Explicit recursion does no longer occur!
- The implementation creates unnecessary intermediate data-structures!

  length could also be implemented as:

$$
\begin{aligned}
\mathsf{length} &= \textbf{let}\ f = \textbf{fun}\ a \to \textbf{fun}\ x \to a+1\\
&\quad\ \textbf{in}\ \mathsf{foldl}\ f\ 0
\end{aligned}
$$

- This implementation avoids to create intermediate lists !!!

Simplification Rules:

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

$$
\begin{aligned}
\mathsf{id} &= \mathbf{fun}\ x\ \to\ x \\
\mathsf{comp} &= \mathbf{fun}\ f\ \to\ \mathbf{fun}\ g\ \to\ \mathbf{fun}\ x\ \to\ f\ (g\ x) \\
\mathsf{comp}_1 &= \mathbf{fun}\ f\ \to\ \mathbf{fun}\ g\ \to\ \mathbf{fun}\ x_1\ \to\ \mathbf{fun}\ x_2\ \to \\
&\qquad f\ (g\ x_1)\ x_2 \\
\mathsf{comp}_2 &= \mathbf{fun}\ f\ \to\ \mathbf{fun}\ g\ \to\ \mathbf{fun}\ x_1\ \to\ \mathbf{fun}\ x_2\ \to \\
&\qquad f\ x_1\ (g\ x_2)
\end{aligned}
$$

Simplification Rules:

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

$$\mathrm{foldl} : (\underline{\alpha \to \beta} \to \alpha) \to \alpha \to \beta\ \mathrm{list} \to \alpha$$

$$
\begin{aligned}
\mathsf{id} &= \mathbf{fun}\ x\ \to\ x \\
\mathsf{comp} &= \mathbf{fun}\ f\ \to\ \mathbf{fun}\ g\ \to\ \mathbf{fun}\ x\ \to\ f\ (g\ x) \\
\mathsf{comp}_1 &= \mathbf{fun}\ f\ \to\ \mathbf{fun}\ g\ \to\ \mathbf{fun}\ x_1\ \to\ \mathbf{fun}\ x_2\ \to \\
&\qquad f\ (g\ x_1)\ x_2 \\
\mathsf{comp}_2 &= \mathbf{fun}\ f\ \to\ \mathbf{fun}\ g\ \to\ \mathbf{fun}\ x_1\ \to\ \mathbf{fun}\ x_2\ \to \\
&\qquad f\ x_1\ (g\ x_2)
\end{aligned}
$$

Simplification Rules:

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

$$
\begin{aligned}
\textsf{id} &= \textbf{fun}\ x \to x \\
\textsf{comp} &= \textbf{fun}\ f \to \textbf{fun}\ g \to \textbf{fun}\ x \to f\ (g\ x) \\
\textsf{comp}_1 &= \textbf{fun}\ f \to \textbf{fun}\ g \to \textbf{fun}\ x_1 \to \textbf{fun}\ x_2 \to \\
&\qquad f\ (g\ x_1)\ x_2 \\
\textsf{comp}_2 &= \textbf{fun}\ f \to \textbf{fun}\ g \to \textbf{fun}\ x_1 \to \textbf{fun}\ x_2 \to \\
&\qquad f\ x_1\ (g\ x_2)
\end{aligned}
$$

comp2 f g = fun x → fun y → f x (g y)

Simplification Rules:

$$
\begin{aligned}
\textsf{comp id}\ f &= \textsf{comp}\ f\ \textsf{id} = f \\
\textsf{comp}_1\ f\ \textsf{id} &= \textsf{comp}_2\ f\ \textsf{id} = f \\
\textsf{map id} &= \textsf{id} \\
\textsf{comp}\ (\textsf{map}\ f)\ (\textsf{map}\ g) &= \textsf{map}\ (\textsf{comp}\ f\ g) \\
\textsf{comp}\ (\textsf{foldl}\ f\ a)\ (\textsf{map}\ g) &= \textsf{foldl}\ (\textsf{comp}_2\ f\ g)\ a \\
\textsf{comp}\ (\textsf{filter}\ p_1)\ (\textsf{filter}\ p_2) &= \textsf{filter}\ (\textbf{fun}\ x \to \ \textbf{if}\ p_2\, x\ \textbf{then}\ p_1\, x \\
&\qquad\qquad \textbf{else}\ \textsf{false}) \\
\textsf{comp}\ (\textsf{foldl}\ f\ a)\ (\textsf{filter}\ p) &= \textbf{let}\ h = \textbf{fun}\ a \to \textbf{fun}\ x \to \ \textbf{if}\ p\, x\ \textbf{then}\ f\, a\, x \\
&\qquad\qquad \textbf{else}\ a \\
&\quad \textbf{in}\ \ \textsf{foldl}\ h\ a
\end{aligned}
$$

Warning:

Function compositions also could occur as nested function calls ...

$$
\begin{aligned}
\textsf{id}\ x &= x \\
\textsf{map id}\ l &= l \\
\textsf{map}\ f\ (\textsf{map}\ g\ l) &= \textsf{map}\ (\textsf{comp}\ f\ g)\ l \\
\textsf{foldl}\ f\ a\ (\textsf{map}\ g\ l) &= \textsf{foldl}\ (\textsf{comp}_2\ f\ g)\ a\, l \\
\textsf{filter}\ p_1\ (\textsf{filter}\ p_2\ l) &= \textsf{filter}\ (\textbf{fun}\ x \to p_1\, x \wedge p_2\, x)\ l \\
\textsf{foldl}\ f\ a\ (\textsf{filter}\ p\ l) &= \textbf{let}\ h = \textbf{fun}\ a \to \textbf{fun}\ x \to \ \textbf{if}\ p\, x\ \textbf{then}\ f\, a\, x \\
&\qquad\qquad \textbf{else}\ a \\
&\quad \textbf{in}\ \ \textsf{foldl}\ h\ a\ l
\end{aligned}
$$

Warning:

Function compositions also could occur as nested function calls ...

$$
\begin{aligned}
\textsf{id}\ x &= x \\
\textsf{map id}\ l &= l \\
\textsf{map}\ f\ (\textsf{map}\ g\ l) &= \textsf{map}\ (\textsf{comp}\ f\ g)\ l \\
\textsf{foldl}\ f\ a\ (\textsf{map}\ g\ l) &= \textsf{foldl}\ (\textsf{comp}_2\ f\ g)\ a\, l \\
\textsf{filter}\ p_1\ (\textsf{filter}\ p_2\ l) &= \textsf{filter}\ (\textbf{fun}\ x \to p_1\, x \wedge p_2\, x)\ l \\
\textsf{foldl}\ f\ a\ (\textsf{filter}\ p\ l) &= \textbf{let}\ h = \textbf{fun}\ a \to \textbf{fun}\ x \to \ \textbf{if}\ p\, x\ \textbf{then}\ f\, a\, x \\
&\qquad\qquad \textbf{else}\ a \\
&\quad \textbf{in}\ \ \textsf{foldl}\ h\ a\ l
\end{aligned}
$$

Example, optimized:

$$
\begin{array}{lcl}
\mathsf{sum} & = & \mathsf{foldl}\ (+)\ 0 \\
\mathsf{length} & = & \mathbf{let}\ f\ =\ \mathsf{comp}_2\ (+)\ (\mathbf{fun}\ x \to 1) \\
 & & \mathbf{in}\ \mathsf{foldl}\ f\ 0 \\
\mathsf{dev} & = & \mathbf{fun}\ l \to \mathbf{let}\ \begin{array}[t]{lcl}
s_1 & = & \mathsf{sum}\ l \\
n & = & \mathsf{length}\ l \\
mean & = & s_1/n \\
f & = & \mathsf{comp}\ (\mathbf{fun}\ x \to x \cdot x) \\
 & & \quad (\mathbf{fun}\ x \to x - mean) \\
g & = & \mathsf{comp}_2\ (+)\ f \\
s_2 & = & \mathsf{foldl}\ g\ 0\ l
\end{array} \\
 & & \quad \mathbf{in}\ s_2/n
\end{array}
$$

$((0 + f\,x_1) + f\,x_2) + \dots$

Remarks:

- All intermediate lists have disappeared :-) ✓
- Only foldl remain — i.e., loops :-)) ✓
- Compositions of functions can be further simplified in the next step by Inlining.
- Inside dev, we then obtain:

$$
\begin{array}{lcl}
g & = & \mathbf{fun}\ a \to \mathbf{fun}\ x \to \mathbf{let}\ \begin{array}[t]{lcl}
x_1 & = & x - mean \\
x_2 & = & x_1 \cdot x_1
\end{array} \\
 & & \qquad\qquad \mathbf{in}\ a + x_2
\end{array}
$$

- The result is a sequence of **let**-definitions !!!

Example, optimized:

$$
\begin{array}{lcl}
\mathsf{sum} & = & \mathsf{foldl}\ (+)\ 0 \\
\mathsf{length} & = & \mathbf{let}\ f\ =\ \mathsf{comp}_2\ (+)\ (\mathbf{fun}\ x \to 1) \\
 & & \mathbf{in}\ \mathsf{foldl}\ f\ 0 \\
\mathsf{dev} & = & \mathbf{fun}\ l \to \mathbf{let}\ \begin{array}[t]{lcl}
s_1 & = & \mathsf{sum}\ l \\
n & = & \mathsf{length}\ l \\
mean & = & s_1/n \\
f & = & \mathsf{comp}\ (\mathbf{fun}\ x \to x \cdot x) \\
 & & \quad (\mathbf{fun}\ x \to x - mean) \\
g & = & \mathsf{comp}_2\ (+)\ f \\
s_2 & = & \mathsf{foldl}\ g\ 0\ l
\end{array} \\
 & & \quad \mathbf{in}\ s_2/n
\end{array}
$$

828

Remarks:

- All intermediate lists have disappeared :-)
- Only foldl remain — i.e., loops :-))
- Compositions of functions can be further simplified in the next step by Inlining.
- Inside dev, we then obtain:

$$
\begin{array}{lcl}
g & = & \mathbf{fun}\ a \to \mathbf{fun}\ x \to \mathbf{let}\ \begin{array}[t]{lcl}
x_1 & = & x - mean \\
x_2 & = & x_1 \cdot x_1
\end{array} \\
 & & \qquad\qquad \mathbf{in}\ a + x_2
\end{array}
$$

- The result is a sequence of **let**-definitions !!!

829

Remarks:

- All intermediate lists have disappeared :-)
- Only foldl remain — i.e., loops :-))
- Compositions of functions can be further simplified in the next step by Inlining.
- Inside dev, we then obtain:

$$
\begin{aligned}
g \;=\; \textbf{fun}\ a \to \textbf{fun}\ x \to \textbf{let}\ \ x_1 &= x - \mathit{mean} \\
x_2 &= x_1 \cdot x_1 \\
\textbf{in}\ \ a + x_2 &
\end{aligned}
$$

- The result is a sequence of **let**-definitions !!!

Extension: Tabulation

If the list has been created by tabulation of a function, the creation of the list sometimes can be avoided ...

$$
\begin{aligned}
\mathsf{tabulate'} \;&=\; \textbf{fun}\ j \to \ \textbf{fun}\ f \to \textbf{fun}\ n \to \\
&\qquad \textbf{if}\ j \geq n\ \textbf{then}\ [\,] \\
&\qquad \textbf{else}\ (f\ j) :: \mathsf{tabulate'}\ (j+1)\ f\ n \\
\mathsf{tabulate} \;&=\; \mathsf{tabulate'}\ 0
\end{aligned}
$$

tabulate f n = [f 0; f 1; ...; f (n-1)]

Then we have:

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

where:

$$
\begin{aligned}
\mathsf{loop'} \;&=\; \textbf{fun}\ j \to \ \textbf{fun}\ f \to \textbf{fun}\ a \to \textbf{fun}\ n \to \\
&\qquad \textbf{if}\ j \geq n\ \textbf{then}\ a \\
&\qquad \textbf{else}\ \mathsf{loop'}\ (j+1)\ f\ (f\ a\ j)\ n \\
\mathsf{loop} \;&=\; \mathsf{loop'}\ 0
\end{aligned}
$$

Extension (2): List Reversals

Sometimes, the ordering of lists or arguments is reversed:

$$
\begin{aligned}
\mathsf{rev'} \;&=\; \textbf{fun}\ a \to \ \textbf{fun}\ l \to \\
&\qquad \textbf{match}\ l\ \textbf{with}\ [\,] \to a \\
&\qquad \mid\ x :: xs \to \mathsf{rev'}\ (x :: a)\ xs \\
\mathsf{rev} \;&=\; \mathsf{rev'}\ [\,] \\
\mathsf{comp\ rev\ rev} \;&=\; \mathsf{id} \\
\mathsf{swap} \;&=\; \textbf{fun}\ f \to \textbf{fun}\ x \to \textbf{fun}\ y \to f\ y\ x \\
\mathsf{comp\ swap\ swap} \;&=\; \mathsf{id}
\end{aligned}
$$

Extension (2): List Reversals

Sometimes, the ordering of lists or arguments is reversed:

$$
\begin{array}{lll}
\mathsf{rev}' & = & \textbf{fun}\ a \to\ \textbf{fun}\ l \to \\
 & & \quad \textbf{match}\ l\ \textbf{with}\ [\,]\ \to\ a \\
 & & \quad \mid\ x :: xs\ \to\ \mathsf{rev}'\ (x :: a)\ xs \\
\mathsf{rev} & = & \mathsf{rev}'\ [\,] \\
\mathsf{comp\ rev\ rev} & = & \mathsf{id} \\
\mathsf{swap} & = & \textbf{fun}\ f\ \to\ \textbf{fun}\ x\ \to\ \textbf{fun}\ y\ \to\ f\ y\ x \\
\mathsf{comp\ swap\ swap} & = & \mathsf{id}
\end{array}
$$

swap f x y ⇒ f y x

Extension (2): List Reversals

Sometimes, the ordering of lists or arguments is reversed:

$$
\begin{array}{lll}
\mathsf{rev}' & = & \textbf{fun}\ a \to\ \textbf{fun}\ l \to \\
 & & \quad \textbf{match}\ l\ \textbf{with}\ [\,]\ \to\ a \\
 & & \quad \mid\ x :: xs\ \to\ \mathsf{rev}'\ (x :: a)\ xs \\
\mathsf{rev} & = & \mathsf{rev}'\ [\,] \\
\mathsf{comp\ rev\ rev} & = & \mathsf{id} \\
\mathsf{swap} & = & \textbf{fun}\ f\ \to\ \textbf{fun}\ x\ \to\ \textbf{fun}\ y\ \to\ f\ y\ x \\
\mathsf{comp\ swap\ swap} & = & \mathsf{id}
\end{array}
$$

832

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

foldr f a (rev xs) =

Discussion:

- The standard implementation of foldr is not tail-recursive.
- The last equation decomposes a foldr into two tail-recursive functions — at the price that an intermediate list is created.
- Therefore, the standard implementation is probably faster :-)
- Sometimes, the operation rev can also be optimized away ...

Extension: Tabulation

If the list has been created by tabulation of a function, the creation of the list sometimes can be avoided ...

$$
\begin{array}{lll}
\mathsf{tabulate}' & = & \textbf{fun}\ j\ \to\ \textbf{fun}\ f\ \to\ \textbf{fun}\ n\ \to \\
 & & \quad \textbf{if}\ j \geq n\ \textbf{then}\ [\,] \\
 & & \quad \textbf{else}\ (f\ j) :: \mathsf{tabulate}'\ (j+1)\ f\ n \\
\mathsf{tabulate} & = & \mathsf{tabulate}'\ 0
\end{array}
$$

$\alpha \to \beta \to \alpha$ $\quad (a \oplus x_1) \oplus x_2$

foldr $\beta \to \alpha \to \alpha$ $\quad (x_n \oplus (x_1 \oplus a))$

$$
\begin{array}{lcl}
\mathsf{id} & = & \mathbf{fun}\ x\ \to\ x \\[2ex]
\mathsf{comp} & = & \mathbf{fun}\ f\ \to\ \mathbf{fun}\ g\ \to\ \mathbf{fun}\ x\ \to\ f\ (g\ x) \\[2ex]
\mathsf{comp}_1 & = & \mathbf{fun}\ f\ \to\ \mathbf{fun}\ g\ \to\ \mathbf{fun}\ x_1\ \to\ \mathbf{fun}\ x_2\ \to \\
 & & \quad f\ (g\ x_1)\ x_2 \\[2ex]
\mathsf{comp}_2 & = & \mathbf{fun}\ f\ \to\ \mathbf{fun}\ g\ \to\ \mathbf{fun}\ x_1\ \to\ \mathbf{fun}\ x_2\ \to \\
 & & \quad f\ x_1\ (g\ x_2)
\end{array}
$$

We have:

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

Here, rev_tabulate tabulates in reverse ordering. This function has properties quite analogous to tabulate:

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

We have:

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

Here, rev_tabulate tabulates in reverse ordering. This function has properties quite analogous to tabulate:

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

834

Extension (3): Dependencies on the Index

- Correctness is proven by induction on the lengthes of occurring lists.
- Similar composition results also hold for transformations which take the current indices into account:

$$
\begin{array}{lcl}
\mathsf{mapi}' & = & \mathbf{fun}\ i\ \to\ \ \mathbf{fun}\ f\ \to\ \mathbf{fun}\ l\ \to\ \mathbf{match}\ l\ \mathbf{with}\ [\,]\ \to\ [\,] \\
 & & \qquad\qquad \mid\ x :: xs\ \to\ f\ i\ x) :: \mathsf{mapi}'\ (i+1)\ f\ xs \\
\mathsf{mapi} & = & \mathsf{mapi}'\ 0
\end{array}
$$