4.4 Application: Inlining

Problem:

- global variables. The program:

$$
\begin{array}{lll}
\textbf{let} & x = 1 & \\
& f = & \textbf{let} \quad x = 2 \\
& & \textbf{in} \quad \textbf{fun}\ y \ \rightarrow\ y + x \\
\textbf{in} & f\ x &
\end{array}
$$

... computes something else than:

$$
\begin{array}{ll}
\textbf{let} & x = 1 \\
& f = \;\; \textbf{let} \quad x = 2 \\
& \qquad\quad \textbf{in} \quad \textbf{fun}\; y \;\to\; y + x \\
\textbf{in} & \boxed{\begin{array}{ll} \textbf{let} & y = x \\ \textbf{in} & y + x \end{array}}
\end{array}
$$

- recursive functions. In the definition:

$$\mathsf{foo} \;=\; \textbf{fun}\; y \;\to\; \mathsf{foo}\; y$$

foo should better not be substituted :-)

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 :-)

Let $D = D[e]$ denote the set of definitions which staticly arrive at e.

- If $e \equiv \textbf{let}\ x_1 = e_1\ \textbf{and} \ldots \textbf{and}\ x_k = e_k\ \textbf{in}\ e_0$ then:

$$\begin{array}{lll} D[e_1] & = & D \\ & \ldots & \\ D[e_k] & = & D \cup \{x_1, \ldots, x_{k-1}\} \\ D[e_0] & = & D \cup \{x_1, \ldots, x_k\} \end{array}$$

- In all other cases, D is propagated to the sub-expressions unchanged :-)

  E.g., if $e \equiv \textbf{fun}\ x \to e_1$ then:

$$D[e_1] = D$$

... in the Example:

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

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

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

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

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

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

$\Longrightarrow$ now we can apply inlining :

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

Removing variable-variable-assignments, we arrive at:

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

Idea 2:

→ We apply our value analysis.

→ We ignore global variables :-)

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

Example: The map-Function

$$
\begin{array}{l}
\textbf{let rec } \mathsf{f} = \textbf{fun } x \to x \cdot x \\
\qquad \mathsf{map} = \textbf{fun } g \to \textbf{fun } x \to \textbf{match } x \\
\qquad\qquad\qquad \textbf{with } [\,] \to [\,] \\
\qquad\qquad\qquad \mid \quad :: z \to \textbf{match } z \textbf{ with } (x_1, x_2) \\
\qquad\qquad\qquad\qquad\qquad \textbf{in } x_1 :: \mathsf{map}\, g\, x_2 \\
\textbf{in } \mathsf{map}\ \mathsf{f}\ \mathit{list}
\end{array}
$$

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

$$
\begin{array}{lll}
\mathsf{h} & = & \textbf{let } g = \boxed{\textbf{fun } x \rightarrow x \cdot x} \\
& & \textbf{in fun } x \rightarrow \textbf{match } x \\
& & \qquad \textbf{with } [\,] \rightarrow [\,] \\
& & \qquad \mid \; :: z \rightarrow \textbf{match } z \textbf{ with } (x_1, x_2) \\
& & \qquad\qquad\qquad \rightarrow g\; x_1 :: \boxed{\mathsf{map}\; g}\; x_2
\end{array}
$$

The inner occurrence of $\mathsf{map}\ g$ can be replaced with h

$\Longrightarrow$ fold-Transformation :-)

$$
\begin{array}{lll}
\mathsf{h} & = & \textbf{let}\ g = \textbf{fun}\ x \ \to\ x \cdot x \\
& & \textbf{in}\ \ \textbf{fun}\ x\ \to\ \textbf{match}\ x \\
& & \qquad\quad \textbf{with}\ \ [\,] \ \to \quad [\,] \\
& & \qquad\qquad \mid \quad\ :: z\ \to\ \textbf{match}\ z\ \textbf{with}\ (x_1, x_2) \\
& & \qquad\qquad\qquad\qquad\qquad\quad \to\ \ g\ x_1 :: \mathsf{h}\ x_2
\end{array}
$$

Inlining the function g yields:

$$
\begin{array}{lll}
\mathsf{h} & = & \textbf{let } g = \textbf{fun } x \to x \cdot x \\
& & \textbf{in } \textbf{fun } x \to \textbf{match } x \\
& & \qquad \textbf{with } [\,] \to [\,] \\
& & \qquad\quad \mid \ \ :: z \to \textbf{match } z \textbf{ with } (x_1, x_2) \\
& & \qquad\qquad\qquad\qquad \to (\,\textbf{let } x = x_1 \\
& & \qquad\qquad\qquad\qquad\quad \textbf{in } x * x\,) :: \mathsf{h}\ x_2
\end{array}
$$

Removing useless definitions and variable-variable assignments yields:

$$
\begin{array}{lll}
\mathsf{h} & = & \textbf{fun}\; x \;\to\; \textbf{match}\; x \\
& & \qquad \textbf{with} \;\; [\,] \;\to\;\; [\,] \\
& & \qquad\quad \mid \;\;\; :: z \;\to\; \textbf{match}\; z\; \textbf{with}\; (x_1, x_2) \\
& & \qquad\qquad\qquad\qquad \to x_1 * x_1 :: \mathsf{h}\; x_2
\end{array}
$$

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:

$$\begin{array}{rcl} \mathsf{map} & = & \textbf{fun}\ f \rightarrow \ \textbf{fun}\ l \rightarrow \textbf{match}\ l\ \textbf{with}\ [\,] \rightarrow \ [\,] \\ & & \qquad\qquad\qquad\quad \mid \ ::\ z \rightarrow (\textbf{match}\ z\ \textbf{with}\ (x, xs) \rightarrow \\ & & \qquad\qquad\qquad\qquad\qquad\qquad f\ x :: \mathsf{map}\ f\ xs) \end{array}$$

$$\begin{aligned}
\mathsf{filter} \quad &= \quad \textbf{fun } p \rightarrow \quad \textbf{fun } l \rightarrow \textbf{match } l \textbf{ with } [\,] \rightarrow [\,] \\
&\qquad\qquad\qquad | \ :: z \rightarrow \big(\textbf{match } z \textbf{ with } (x, xs) \rightarrow \\
&\qquad\qquad\qquad\qquad\qquad \textbf{if } p\,x \textbf{ then } x :: \mathsf{filter}\ p\ xs \\
&\qquad\qquad\qquad\qquad\qquad \textbf{else } \mathsf{filter}\ p\ xs\big)
\end{aligned}$$

$$\begin{aligned}
\mathsf{foldl} \quad &= \quad \textbf{fun } f \rightarrow \quad \textbf{fun } a \rightarrow \textbf{fun } l \rightarrow \textbf{match } l \textbf{ with } [\,] \rightarrow a \\
&\qquad\qquad\qquad | \ :: z \rightarrow \big(\textbf{match } z \textbf{ with } (x, xs) \rightarrow \\
&\qquad\qquad\qquad\qquad\qquad \mathsf{foldl}\ f\ (f\ a\ x)\ xs\big)
\end{aligned}$$

$$\mathsf{id} \quad = \quad \mathbf{fun}\ x \ \rightarrow\ x$$

$$\mathsf{comp} \quad = \quad \mathbf{fun}\ f \ \rightarrow\ \mathbf{fun}\ g \ \rightarrow\ \mathbf{fun}\ x \ \rightarrow\ f\ (g\ x)$$

$$\begin{aligned} \mathsf{comp}_1 \quad = \quad & \mathbf{fun}\ f \ \rightarrow\ \mathbf{fun}\ g \ \rightarrow\ \mathbf{fun}\ x_1 \ \rightarrow\ \mathbf{fun}\ x_2 \ \rightarrow \\ & \quad f\ (g\ x_1)\ x_2 \end{aligned}$$

$$\begin{aligned} \mathsf{comp}_2 \quad = \quad & \mathbf{fun}\ f \ \rightarrow\ \mathbf{fun}\ g \ \rightarrow\ \mathbf{fun}\ x_1 \ \rightarrow\ \mathbf{fun}\ x_2 \ \rightarrow \\ & \quad f\ x_1\ (g\ x_2) \end{aligned}$$

Example:

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

Observations:

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

  length could also be implemented as:

$$\begin{array}{lllll} \text{length} & = & \textbf{let}\ f & = & \textbf{fun}\ a \rightarrow \textbf{fun}\ x \rightarrow a + 1 \\ & & \textbf{in}\ \ \text{foldl}\ f\ 0 & & \end{array}$$

- This implementation avoids to create intermediate lists !!!

Simplification Rules:

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

Simplification Rules:

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

Warning:

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

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

Example, optimized:

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

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:

$$
\begin{array}{lllllll}
g & = & \textbf{fun}\ a \ \rightarrow\ \textbf{fun}\ x \ \rightarrow\ \textbf{let} & x_1 & = & x - \mathit{mean} \\
 & & & x_2 & = & x_1 \cdot x_1 \\
 & & \textbf{in}\ \ a + x_2 & & &
\end{array}
$$

- 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 ...

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

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:

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

Extension (2): List Reversals

Sometimes, the ordering of lists or arguments is reversed:

$$
\begin{array}{lcl}
\mathsf{rev}' & = & \mathbf{fun}\, a \rightarrow \ \mathbf{fun}\, l \rightarrow \\
 & & \qquad\qquad \mathbf{match}\ l\ \mathbf{with}\ [\,] \ \rightarrow \ a \\
 & & \qquad\qquad \mid \ \ ::z \ \rightarrow \ \ (\mathbf{match}\ z\ \mathbf{with}\ (x, xs) \ \rightarrow \\
 & & \qquad\qquad\qquad\qquad \mathsf{rev}'\ (x :: a)\ xs) \\
\mathsf{rev} & = & \mathsf{rev}'\ [\,] \\
 & & \\
\mathsf{comp\ rev\ rev} & = & \mathsf{id} \\
 & & \\
\mathsf{swap} & = & \mathbf{fun}\ f \ \rightarrow \ \mathbf{fun}\ x \ \rightarrow \ \mathbf{fun}\ y \ \rightarrow \ f\ y\ x \\
 & & \\
\mathsf{comp\ swap\ swap} & = & \mathsf{id}
\end{array}
$$

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

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 ...

We have:

$$\begin{aligned}
\mathsf{comp}\ \mathsf{rev}\ (\mathsf{map}\ f) &= \mathsf{comp}\ (\mathsf{map}\ f)\ \mathsf{rev} \\
\mathsf{comp}\ \mathsf{rev}\ (\mathsf{filter}\ p) &= \mathsf{comp}\ (\mathsf{filter}\ p)\ \mathsf{rev} \\
\mathsf{comp}\ \mathsf{rev}\ (\mathsf{tabulate}\ f) &= \mathsf{rev_tabulate}\ f
\end{aligned}$$

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

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

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:

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

Analogously, there is index-dependent accumulation:

$$\begin{array}{lcl}
\text{foldli}' & = & \textbf{fun}\ i\ \to\ \textbf{fun}\ f\ \to\ \textbf{fun}\ a\ \to\ \textbf{fun}\ l\ \to \\
& & \qquad \textbf{match}\ l\ \textbf{with}\ [\,]\ \to\ a \\
& & \qquad \mid\ ::\ z\ \to\ (\textbf{match}\ z\ \textbf{with}\ (x, xs)\ \to \\
& & \qquad\qquad \text{foldli}'\ (i+1)\ f\ (f\ i\ a\ x)\ xs) \\
\text{foldli} & = & \text{foldli}'\ 0
\end{array}$$

For composition, we must take care that always the same indices are used. This is achieved by:

$$\mathsf{compi} \;=\; \mathbf{fun}\ f \;\rightarrow\; \mathbf{fun}\ g \;\rightarrow\; \mathbf{fun}\ i \;\rightarrow\; \mathbf{fun}\ x \;\rightarrow\; f\ i\ (g\ i\ x)$$

$$\begin{aligned}\mathsf{compi}_1 \;=\; & \mathbf{fun}\ f \;\rightarrow\; \mathbf{fun}\ g \;\rightarrow\; \mathbf{fun}\ i \;\rightarrow\; \mathbf{fun}\ x_1 \;\rightarrow\; \mathbf{fun}\ x_2 \;\rightarrow \\ & \quad f\ i\ (g\ i\ x_1)\ x_2\end{aligned}$$

$$\begin{aligned}\mathsf{compi}_2 \;=\; & \mathbf{fun}\ f \;\rightarrow\; \mathbf{fun}\ g \;\rightarrow\; \mathbf{fun}\ i \;\rightarrow\; \mathbf{fun}\ x_1 \;\rightarrow\; \mathbf{fun}\ x_2 \;\rightarrow \\ & \quad f\ i\ x_1\ (g\ i\ x_2)\end{aligned}$$

$$\begin{aligned}\mathsf{cmp}_1 \;=\; & \mathbf{fun}\ f \;\rightarrow\; \mathbf{fun}\ g \;\rightarrow\; \mathbf{fun}\ i \;\rightarrow\; \mathbf{fun}\ x_1 \;\rightarrow\; \mathbf{fun}\ x_2 \;\rightarrow \\ & \quad f\ i\ x_1\ (g\ x_2)\end{aligned}$$

$$\begin{aligned}\mathsf{cmp}_2 \;=\; & \mathbf{fun}\ f \;\rightarrow\; \mathbf{fun}\ g \;\rightarrow\; \mathbf{fun}\ i \;\rightarrow\; \mathbf{fun}\ x_1 \;\rightarrow\; \mathbf{fun}\ x_2 \;\rightarrow \\ & \quad f\ x_1\ (g\ i\ x_2)\end{aligned}$$

Then:

$$
\begin{array}{lcl}
\mathsf{comp}\,(\mathsf{mapi}\ f)\ (\mathsf{map}\ g) & = & \mathsf{mapi}\,(\mathsf{comp}_2\ f\ g) \\
\mathsf{comp}\,(\mathsf{map}\ f)\ (\mathsf{mapi}\ g) & = & \mathsf{mapi}\,(\mathsf{comp}\ f\ g) \\
\mathsf{comp}\,(\mathsf{mapi}\ f)\ (\mathsf{mapi}\ g) & = & \mathsf{mapi}\,(\mathsf{compi}\ f\ g) \\
\mathsf{comp}\,(\mathsf{foldli}\ f\ a)\ (\mathsf{map}\ g) & = & \mathsf{foldli}\,(\mathsf{cmp}_1\ f\ g)\ a \\
\mathsf{comp}\,(\mathsf{foldl}\ f\ a)\ (\mathsf{mapi}\ g) & = & \mathsf{foldli}\,(\mathsf{cmp}_2\ f\ g)\ a \\
\mathsf{comp}\,(\mathsf{foldli}\ f\ a)\ (\mathsf{mapi}\ g) & = & \mathsf{foldli}\,(\mathsf{compi}_2\ f\ g)\ a \\
\mathsf{comp}\,(\mathsf{foldli}\ f\ a)\ (\mathsf{tabulate}\ g) & = & \textbf{let}\ h = \ \textbf{fun}\ a\ \rightarrow\ \textbf{fun}\ i\ \rightarrow \\
 & & \qquad\qquad f\ i\ a\ (g\ i) \\
 & & \textbf{in}\ \ \mathsf{loop}\ h\ a
\end{array}
$$

Discussion:

- Warning: index-dependent transformations may not commute with rev or filter.
- All our rules can only be applied if the functions id, map, mapi, foldl, foldli, filter, rev, tabulate, rev_tabulate, loop, rev_loop, ... are provided by a standard library: Only then the algebraic properties can be guaranteed !!!
- Similar simplification rules can be derived for any kind of tree-like data-structure tree α .
- These also provide operations map, mapi and foldl, foldli with corresponding rules.
- Further opportunities are opened up by functions to_list and from_list ...