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:

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

- recursive functions. In the definition:

$$\mathsf{foo} \ = \ \textbf{fun}\ y \ \rightarrow \ \mathsf{foo}\ y$$

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:

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

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

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

$$D[e_1] = D$$

... in the Example:

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

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

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

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

$\Longrightarrow$ 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 :

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

Removing variable-variable-assignments, we arrive at:

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

Idea 2:

→ We apply our value analysis.

→ We ignore global variables :-)

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

Example: The map-Function

```
let rec f = fun x → x · x
        map = fun g → fun x → match x
                              with [ ] → [ ]
                                 | :: z → match z with (x1, x2)
                                          in x1 :: map g x2
in map f list
```

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

```
h = let g = fun x → x · x
    in fun x → match x
                 with [] → []
                    | :: z → match z with (x1, x2)
                             → g x1 :: map g x2
```

The inner occurrence of map g can be replaced with h

$\Longrightarrow$ fold-Transformation :-)

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

Inlining the function g yields:

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

Removing useless definitions and variable-variable assignments yields:

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

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:

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

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

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

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

Example:

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

Observations:

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

  length could also be implemented as:

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

- This implementation avoids to create intermediate lists !!!

Simplification Rules:

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

Simplification Rules:

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

Warning:

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

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

Example, optimized:

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

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:

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

- 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 ...

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

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:

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

Extension (2): List Reversals

Sometimes, the ordering of lists or arguments is reversed:

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

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

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:

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

Analogously, there is index-dependent accumulation:

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

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

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

$$\mathsf{compi}_1 = \mathbf{fun}\ f \to \mathbf{fun}\ g \to \mathbf{fun}\ i \to \mathbf{fun}\ x_1 \to \mathbf{fun}\ x_2 \to f\ i\ (g\ i\ x_1)\ x_2$$

$$\mathsf{compi}_2 = \mathbf{fun}\ f \to \mathbf{fun}\ g \to \mathbf{fun}\ i \to \mathbf{fun}\ x_1 \to \mathbf{fun}\ x_2 \to f\ i\ x_1\ (g\ i\ x_2)$$

$$\mathsf{cmp}_1 = \mathbf{fun}\ f \to \mathbf{fun}\ g \to \mathbf{fun}\ i \to \mathbf{fun}\ x_1 \to \mathbf{fun}\ x_2 \to f\ i\ x_1\ (g\ x_2)$$

$$\mathsf{cmp}_2 = \mathbf{fun}\ f \to \mathbf{fun}\ g \to \mathbf{fun}\ i \to \mathbf{fun}\ x_1 \to \mathbf{fun}\ x_2 \to f\ x_1\ (g\ i\ x_2)$$

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 ...