$$\frac{(e_1, \eta) \Longrightarrow v_1 \quad \ldots \quad (e_k, \eta) \Longrightarrow v_k}{((e_1, \ldots, e_k), \eta) \Longrightarrow (v_1, \ldots, v_k)}$$

Global Definition:

$$\frac{\textbf{let rec } \ldots\ x = e\ \ldots\ \textbf{in } \ldots \qquad (e, \emptyset) \Longrightarrow v}{(x, \eta) \Longrightarrow v}$$

Function Application:

$$\frac{(e_1, \eta) \Longrightarrow (\textbf{fun}\ x\ \rightarrow\ e, \eta_1) \quad (e_2, \eta) \Longrightarrow v_2 \quad (e, \eta_1 \oplus \{x \mapsto v_2\}) \Longrightarrow v_3}{(e_1\ e_2, \eta) \Longrightarrow v_3}$$

Case Distinction 1:

$$\frac{(e, \eta) \Longrightarrow b \qquad (e_i, \eta) \Longrightarrow v}{(\textbf{match}\ e\ \textbf{with}\ p_1 \rightarrow e_1 \mid \ldots \mid p_k \rightarrow e_k, \eta) \Longrightarrow v}$$

if $p_i \equiv b$ is the first pattern which matches b :-)

Case Distinction 2:

$$\frac{(e, \eta) \Longrightarrow c\, v_1 \ldots v_k \qquad (e_i, \eta \oplus \{z_1 \mapsto v_1, \ldots, z_k \mapsto v_k\}) \Longrightarrow v}{(\textbf{match}\ e\ \textbf{with}\ p_1\ \rightarrow\ e_1 \mid \ldots \mid p_k\ \rightarrow\ e_k, \eta) \Longrightarrow v}$$

if $p_i \equiv c\ z_1 \ldots z_k$ is the first pattern which matches $c\, v_1 \ldots v_k$:-)

Case Distinction 3:

$$\frac{(e, \eta) \Longrightarrow (v_1, \ldots, v_k) \qquad (e_i, \eta \oplus \{y_1 \mapsto v_1, \ldots, y_1 \mapsto v_k\}) \Longrightarrow v}{(\textbf{match}\ e\ \textbf{with}\ p_1\ \rightarrow\ e_1 \mid \ldots \mid p_k\ \rightarrow\ e_k, \eta) \Longrightarrow v}$$

if $p_i \equiv (y_1, \ldots, y_k)$ is the first pattern which matches $(v_1, \ldots, v_k)$

:-)

Case Distinction 4:

$$\frac{(e, \eta) \Longrightarrow v' \qquad (e_i, \eta \oplus \{x \mapsto v'\}) \Longrightarrow v}{(\textbf{match}\ e\ \textbf{with}\ p_1\ \rightarrow\ e_1 \mid \ldots \mid p_k\ \rightarrow\ e_k, \eta) \Longrightarrow v}$$

if $p_i \equiv x$ is the first pattern which matches v' :-)

Local Definitions:

$$\frac{(e_1, \eta) \Longrightarrow v_1 \qquad (e_0, \eta \oplus \{x_1 \mapsto v_1\}) \Longrightarrow v_0}{(\textbf{let } x_1 = e_1 \textbf{ in } e_0, \eta) \Longrightarrow v_0}$$

Variables:

$$(x, \eta) \Longrightarrow \eta(x)$$

Correctness of the Analysis:

For every (e, η) occurring in a proof for the program, it should hold:

- If $\eta\,(x) = v$, then $[v] \,\Delta\, \mathcal{L}(x)$.
- If $(e, \eta) \Longrightarrow v$, then $[v] \,\Delta\, \mathcal{L}(e)$ ...

- where $[v]$ is the stripped expression corresponding to v, i.e., obtained by removing all environments, and
- $v \,\Delta\, L$ iff $v \in L$ or L has an expression v' which evaluates to v.

Conclusion:

$\mathcal{L}(e)$ returns a superset of the values to which e is evaluated :-)

4.4 Application: Inlining

Problem:

- global variables. The program:

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

... computes something else than:

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

- 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 $e \equiv \textbf{let}\ x_1 = e_1\ \textbf{in}\ e_0$ then:

$$\begin{aligned} D[e_1] &= D \\ D[e_0] &= D \cup \{x_1\} \end{aligned}$$

- If $e \equiv \textbf{fun}\, x \rightarrow e_1$ then:

$$D[e_1] = D \cup \{x\}$$

- Similarly, for $e \equiv \textbf{match} \ldots c\, x_1 \ldots x_k \rightarrow e_i \ldots$,

$$D[e_i] = D \cup \{x_1, \ldots, x_k\}$$

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

... in the Example:

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

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

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

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

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

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

$\Longrightarrow$ now we can apply inlining :

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

Removing variable-variable-assignments, we arrive at:

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

Idea 2:

- $\rightarrow$ We apply our value analysis.
- $\rightarrow$ We ignore global variables :-)
- $\rightarrow$ We only substitute functions without free variables :-))

Example: The map-Function

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

- 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:

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

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

$\Longrightarrow$ fold-Transformation :-)

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

Inlining the function g yields:

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

Removing useless definitions and variable-variable assignments yields:

$$
\begin{array}{lcl}
\mathsf{h} & = & \mathbf{fun}\ x\ \rightarrow\ \mathbf{match}\ x \\
 & & \quad \mathbf{with}\ \ [\,]\ \rightarrow\ \ \ [\,] \\
 & & \quad\ \ \mid\ \ x{::}xs\ \rightarrow\ x * x :: \mathsf{h}\ xs
\end{array}
$$

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:

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

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

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

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

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

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

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

Example:

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

Observations:

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

  length could also be implemented as:

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

- 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:

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

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:

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

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 ...

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

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:

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

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

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 ...