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:

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

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

:-)

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 $\quad p_i \equiv (y_1, \ldots, y_k) \quad$ 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:

$$\frac{\begin{array}{l}(e_1, \eta) \Longrightarrow v_1 \\ (e_2, \eta \oplus \{x_1 \mapsto v_1\}) \Longrightarrow v_2 \\ \ldots \\ (e_k, \eta \oplus \{x_1 \mapsto v_1, \ldots, x_{k-1} \mapsto v_{k-1}\}) \Longrightarrow v_k \\ (e_0, \eta \oplus \{x_1 \mapsto v_1, \ldots, x_k \mapsto v_k\}) \Longrightarrow v_0\end{array}}{(\textbf{let } x_1 = e_1 \textbf{ and} \ldots \textbf{and } x_k = e_k \textbf{ in } e_0, \eta) \Longrightarrow v_0}$$

Correctness of the Analysis:

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

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

- where $[v]$ is the stripped expression corresponding to v, i.e., obtained by removing all environments.

Conclusion:

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

4.4 Application: Inlining

Problem:

- global variables. The program:

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

... computes something else than:

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

- 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:

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

item 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:

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

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

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

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

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

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

$\Longrightarrow$ now we can apply inlining :

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

Removing variable-variable-assignments, we arrive at:

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

Idea 2:

$\longrightarrow$ We apply our value analysis.

$\longrightarrow$ We ignore global variables :-)

$\longrightarrow$ We only substitute functions without free variables :-))

Example: The map-Function

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

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

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

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

$\Longrightarrow$ fold-Transformation :-)

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

Inlining the function g yields:

$$
\begin{array}{lllll}
\mathsf{h} & = & \multicolumn{3}{l}{\textbf{let}\ g = \textbf{fun}\ x \ \to\ x \cdot x} \\
 & & \multicolumn{3}{l}{\textbf{in}\ \ \textbf{fun}\ x\ \to\ \textbf{match}\ x} \\
 & & & \textbf{with}\ \ [\,] \ \to & [\,] \\
 & & & \mid \quad x{::}xs\ \to & (\,\textbf{let}\ x = x \\
 & & & & \quad \textbf{in}\ \ x * x\,)\ ::\ \mathsf{h}\ xs
\end{array}
$$

Removing useless definitions and variable-variable assignments yields:

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

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:

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

$$\begin{aligned}\mathsf{filter} \;&=\; \mathbf{fun}\ p \;\to\; \mathbf{fun}\ l \;\to\; \mathbf{match}\ l\ \mathbf{with}\ [\,] \;\to\; [\,]\\ &\qquad\qquad\qquad\quad \mid\ x{::}xs \;\to\; \mathbf{if}\ p\,x\ \mathbf{then}\ x :: \mathsf{filter}\ p\ xs\\ &\qquad\qquad\qquad\qquad\qquad\qquad\ \mathbf{else}\ \mathsf{filter}\ p\ xs)\end{aligned}$$

$$\begin{aligned}\mathsf{foldl} \;&=\; \mathbf{fun}\ f \;\to\; \mathbf{fun}\ a \;\to\; \mathbf{fun}\ l \;\to\; \mathbf{match}\ l\ \mathbf{with}\ [\,] \;\to\; a\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid\ x{::}xs \;\to\; \mathsf{foldl}\ f\ (f\,a\,x)\ xs)\end{aligned}$$

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

Example:

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

Observations:

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

  length could also be implemented as:

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

- 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:

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

Warning:

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

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

Example, optimized:

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

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:

$$
\begin{array}{lllllllll}
g & = & \mathbf{fun}\ a & \to & \mathbf{fun}\ x & \to & \mathbf{let} & x_1 & = & x - \mathit{mean} \\
 & & & & & & & x_2 & = & x_1 \cdot x_1 \\
 & & & & & & \mathbf{in} & a + x_2 & &
\end{array}
$$

- 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 ...

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

Then we have:

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

where:

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

Extension (2): List Reversals

Sometimes, the ordering of lists or arguments is reversed:

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