Script generated by TTT

Title: Seidl: Programmoptimierung (18.01.2012)

Date: Wed Jan 18 12:30:26 CET 2012

Duration: 90:51 min

Pages: 57

Expressions are evaluated w.r.t. an environment $\eta : Vars \to Values$.

The Big-Step operational semantics provides rules to infer the value to which an expression is evaluated w.r.t. a given environment, i.e., deals with statements of the form:

$$(e, \eta) \Longrightarrow v$$

Values:

$$(b, \eta) \Longrightarrow b$$

$$(\mathbf{fun}\, x \to e, \eta) \Longrightarrow (\mathbf{fun}\, x \to e, \eta)$$

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

Operator applications are treated analogously!

$$
\begin{array}{lcl}
\ldots & & \\
[\![h]\!]^\sharp & = & \{1, 2\} \\
[\![t]\!]^\sharp & = & \{[2], []\} \\
[\![\mathsf{app}\, t]\!]^\sharp & = & \\
[\![\mathsf{app}\, [1;2]]\!]^\sharp & = & \{\mathbf{fun}\, y \to y, \mathbf{fun}\, y \to h :: \mathsf{app}\ldots\} \\
[\![\mathsf{app}\, t\, y]\!]^\sharp & = & \\
[\![\mathsf{app}\, [1;2]\, [3]]\!]^\sharp & = & \{[3], h :: \mathsf{app}\ldots\}
\end{array}
$$

Values $c\, e_1 \ldots e_k$, $(e_1, \ldots, e_k)$ or operator applications $e_1 \Box e_2$ now are interpreted as recursive calls $c\, [\![e_1]\!]^\sharp \ldots [\![e_k]\!]^\sharp$, $([\![e_1]\!]^\sharp, \ldots, [\![e_k]\!]^\sharp)$ or $[\![e_1]\!]^\sharp \Box [\![e_2]\!]^\sharp$, respectively.

$\Longrightarrow$ regular tree grammar

... in the Example:

We obtain for $A = [\![\mathsf{app}\, t\, y]\!]^\sharp$:

$$
\begin{array}{lcl}
A & \to & [3] \;\mid\; [\![h]\!]^\sharp :: A \\
[\![h]\!]^\sharp & \to & 1 \;\mid\; 2
\end{array}
$$

Let $\mathcal{L}(e)$ denote the set of terms derivable from $[\![e]\!]^\sharp$ w.r.t. the regular tree grammar. Thus, e.g.,

$$
\begin{array}{lcl}
\mathcal{L}(h) & = & \{1, 2\} \\
\mathcal{L}(\mathsf{app}\, t\, y) & = & \{[a_1; \ldots, a_r; 3] \mid r \geq 0, a_i \in \{1, 2\}\}
\end{array}
$$

4.3 An Operational Semantics

Idea:

We construct a Big-Step operational semantics which evaluates expressions w.r.t. an environment :-)

Values are of the form:

$$v ::= b \mid c\, v_1 \ldots c_k \mid (v_1, \ldots, v_k) \mid (\mathbf{fun}\, x \to e, \eta)$$

Examples for Values:

$c\,1$

$[1;2] = ::\ 1\ (::\ 2\ [\,])$

$(\mathbf{fun}\, x \to x::y, \{y \mapsto [5]\})$

Expressions are evaluated w.r.t. an environment $\eta : Vars \to Values$.

The Big-Step operational semantics provides rules to infer the value to which an expression is evaluated w.r.t. a given environment, i.e., deals with statements of the form:

$$(e, \eta) \Longrightarrow v$$

Values:

$$(b, \eta) \Longrightarrow b$$

$$(\mathbf{fun}\, x \to e, \eta) \Longrightarrow (\mathbf{fun}\, x \to e, \eta)$$

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

Operator applications are treated analogously!

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

Global Definition:

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

Function Application:

$$\frac{(e_1, \eta) \Longrightarrow (\mathbf{fun}\, x \to e, \eta_1) \qquad (e_2, \eta) \Longrightarrow v_2 \qquad (e, \eta_1 \oplus \{x \mapsto v_2\}) \Longrightarrow v_3}{(e_1\ e_2, \eta) \Longrightarrow v_3}$$

Case Distinction 1:

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

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

Case Distinction 3:

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

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

Case Distinction 4:

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

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

Local Definitions:

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

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 :-)

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:

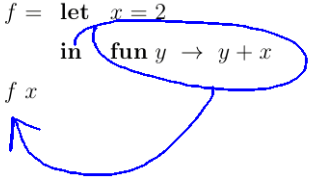
$\mathcal{L}(e)$ returns a superset of the values to which e is evaluated :-)

4.4 Application: Inlining

Problem:

- global variables. The program:

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



... computes something else than:

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

- recursive functions. In the definition:

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

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 :-)

4.4 Application: Inlining

Problem:

- global variables. The program:

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

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 :-)

808

4.4 Application: Inlining

Problem:

- global variables. The program:

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

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

... in the Example:

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

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

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

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:

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

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

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

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

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

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

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

```
    let  x = 1
in let  x1 = 2
in let  f = fun y → y + x1
in      let  y = x
        in   y + x1
```

Removing variable-variable-assignments, we arrive at:

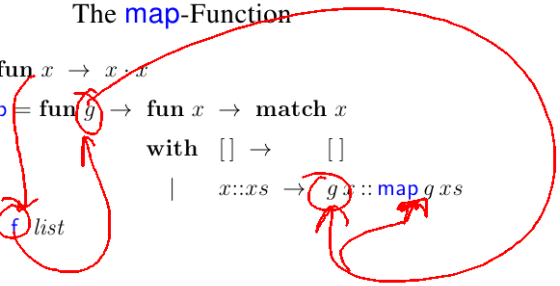

```
    let  x = 1
in let  x1 = 2
in let  f = fun y → y + x1
in      x + x1
```


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
  and    map = fun g → fun x → match x
                       with [] →      []
                          | x::xs → g x :: map g xs
in  map f list
```



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
  and    map = fun g → fun x → match x
                       with [] →      []
                          | x::xs → g x :: map g xs
in  map f list
```

- 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:

```
h = let g = fun x → x · x
    in fun x → match x
         with [] →      []
            | x::xs →  g x :: map g xs
```


The inner occurrence of `map g` can be replaced with `h`

$\Longrightarrow$ fold-Transformation :-)

```
h = let g = fun x → x · x
    in fun x → match x
         with [] →      []
            | x::xs →  g x :: h xs
```


Inlining the function g yields:

```
h = let g = fun x → x · x
    in fun x → match x
         with [] →      []
            | x::xs →  (let x = x
                        in x ∗ x) :: h xs
```


818

Inlining the function g yields:

```
h = let g = fun x → x · x
    in fun x → match x
         with [] →      []
            | x::xs →  (let x = x
                        in x ∗ x) :: h xs
```

818

Removing useless definitions and variable-variable assignments yields:

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

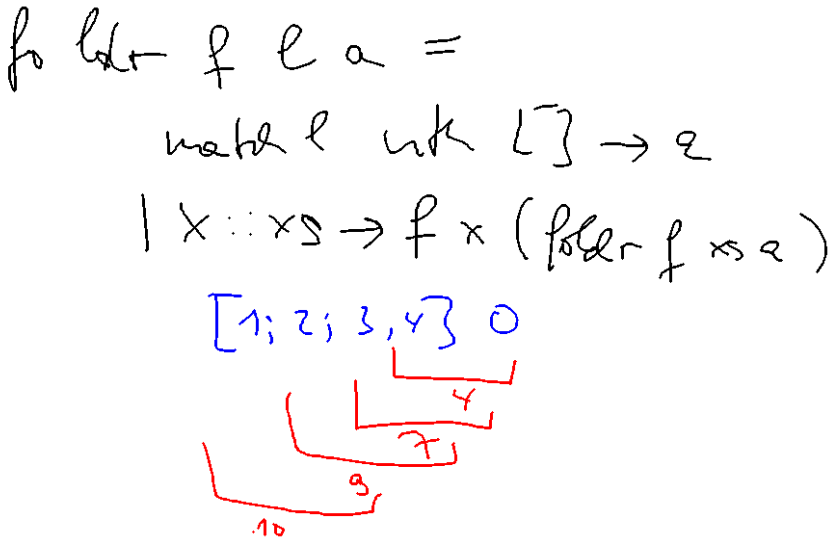
4.5 Deforestation

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

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

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

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



foldl (::) [] list = rev list

foldr

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

Example:

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

Observations:

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

  length could also be implemented as:

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

- This implementation avoids to create intermediate lists !!!

Example:

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

Observations:

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

  length could also be implemented as:

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

- This implementation avoids to create intermediate lists !!!

Simplification Rules:

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

Simplification Rules:

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

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 ...

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

Warning:

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

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

827

Warning:

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

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

827

Example, optimized:

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

Example:

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

Example, optimized:

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

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:

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

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

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:

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

- 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 ...

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

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

Then we have:

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

where:

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

Extension (2): List Reversals

Sometimes, the ordering of lists or arguments is reversed:

$$\begin{array}{lcl} \mathsf{rev'} & = & \textbf{fun}\ a \to \textbf{fun}\ l \to \\ & & \qquad \textbf{match}\ l\ \textbf{with}\ [\,] \to\ a \\ & & \qquad \mid\ x :: xs \to\ \mathsf{rev'}\ (x :: a)\ xs \\ \mathsf{rev} & = & \mathsf{rev'}\ [\,] \end{array}$$

$$\mathsf{comp\ rev\ rev} = \mathsf{id}$$

$$\mathsf{swap} = \textbf{fun}\ f \to \textbf{fun}\ x \to \textbf{fun}\ y \to f\ y\ x$$

$$\mathsf{comp\ swap\ swap} = \mathsf{id}$$

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

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:

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