Script generated by TTT

Title: Seidl: Programmoptimierung (03.02.2014)

Date: Mon Feb 03 14:15:29 CET 2014

Duration: 91:03 min

Pages: 60

- int-values returned by operators are described by the unevaluated expression;

  Operator applications might return Boolean values or other basic values. Therefore, we do replace tests for basic values by non-deterministic choice ...
- Assume $e \equiv \mathbf{match}\ e_0\ \mathbf{with}\ p_1 \to e_1 \mid \ldots \mid p_k \to e_k$.
  Then we generate for $p_i \equiv b$ (basic value),

$$[\![e]\!]^\sharp \supseteq [\![e_i]\!]^\sharp \ : \ \emptyset$$

...

If $p_i \equiv c\, y_1 \ldots y_k$ and $v \equiv c\, e'_1 \ldots e'_k$ is a value, then

$$\begin{array}{lll} [\![e]\!]^\sharp & \supseteq & (v \in [\![e_0]\!]^\sharp)\,?\,[\![e_i]\!]^\sharp \ : \ \emptyset \\ [\![y_j]\!]^\sharp & \supseteq & (v \in [\![e_0]\!]^\sharp)\,?\,[\![e'_j]\!]^\sharp \ : \ \emptyset \end{array}$$

If $p_i \equiv (y_1, \ldots, y_k)$ and $v \equiv (e'_1, \ldots, e'_k)$ is a value, then

$$\begin{array}{lll} [\![e]\!]^\sharp & \supseteq & (v \in [\![e_0]\!]^\sharp)\,?\,[\![e_i]\!]^\sharp \ : \ \emptyset \\ [\![y_j]\!]^\sharp & \supseteq & (v \in [\![e_0]\!]^\sharp)\,?\,[\![e'_j]\!]^\sharp \ : \ \emptyset \end{array}$$

If $p_i \equiv y$, then

$$\begin{array}{lll} [\![e]\!]^\sharp & \supseteq & [\![e_i]\!]^\sharp \\ [\![y]\!]^\sharp & \supseteq & [\![e_0]\!]^\sharp \end{array}$$

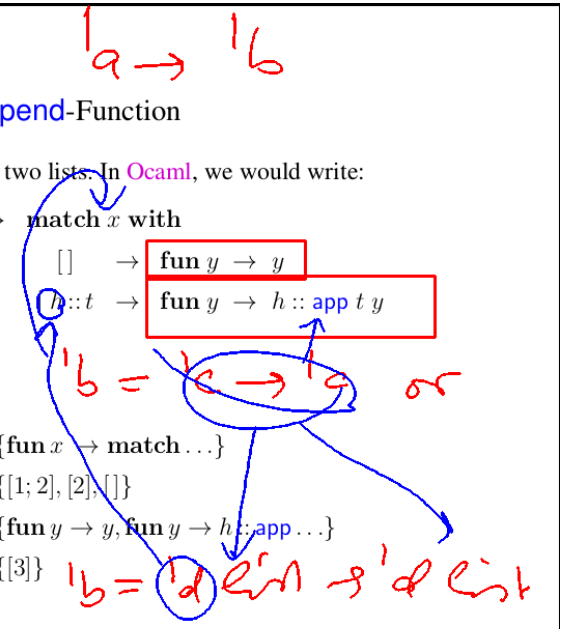
Example The append-Function

Consider the concatenation of two lists. In Ocaml, we would write:

$$\begin{array}{l} \mathbf{let\ rec}\ \mathsf{app} \ = \ \mathbf{fun}\ x \ \to \ \mathbf{match}\ x\ \mathbf{with} \\ \qquad\qquad [\,] \ \to \ \mathbf{fun}\ y \ \to \ y \\ \qquad\qquad h :: t \ \to \ \mathbf{fun}\ y \ \to \ h :: \mathsf{app}\ t\ y \\ \mathbf{in}\ \mathsf{app}\ [1;2]\ [3] \end{array}$$

The analysis then results in:

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



Example The append-Function

Consider the concatenation of two lists. In Ocaml, we would write:

$$\begin{array}{l}\textbf{let rec } \mathsf{app} \;=\; \textbf{fun } x \;\rightarrow\; \textbf{match } x \textbf{ with}\\ \qquad\qquad\qquad\qquad [\,] \;\rightarrow\; \textbf{fun } y \;\rightarrow\; y\\ \qquad\qquad\qquad\qquad \mid h :: t \;\rightarrow\; \textbf{fun } y \;\rightarrow\; h :: \mathsf{app}\ t\ y\\ \textbf{in } \mathsf{app}\ [1;2]\ [3]\end{array}$$

The analysis then results in:

$$\begin{array}{lcl}[\![\mathsf{app}]\!]^\sharp & = & \{\textbf{fun } x \rightarrow \textbf{match} \ldots\}\\ [\![x]\!]^\sharp & = & \{[1;2],[2],[\,]\}\\ [\![\textbf{match} \ldots]\!]^\sharp & = & \{\textbf{fun } y \rightarrow y, \textbf{fun } y \rightarrow h :: \mathsf{app} \ldots\}\\ [\![y]\!]^\sharp & = & \{[3]\}\\ \ldots\end{array}$$

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

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

Example The append-Function

Consider the concatenation of two lists. In Ocaml, we would write:

$$\begin{array}{l}\textbf{let rec } \mathsf{app} \;=\; \textbf{fun } x \;\rightarrow\; \textbf{match } x \textbf{ with}\\ \qquad\qquad\qquad\qquad [\,] \;\rightarrow\; \textbf{fun } y \;\rightarrow\; y\\ \qquad\qquad\qquad\qquad \mid h :: t \;\rightarrow\; \textbf{fun } y \;\rightarrow\; h :: \mathsf{app}\ t\ y\\ \textbf{in } \mathsf{app}\ [1;2]\ [3]\end{array}$$

The analysis then results in:

$$\begin{array}{lcl}[\![\mathsf{app}]\!]^\sharp & = & \{\textbf{fun } x \rightarrow \textbf{match} \ldots\}\\ [\![x]\!]^\sharp & = & \{[1;2],[2],[\,]\}\\ [\![\textbf{match} \ldots]\!]^\sharp & = & \{\textbf{fun } y \rightarrow y, \textbf{fun } y \rightarrow h :: \mathsf{app} \ldots\}\\ [\![y]\!]^\sharp & = & \{[3]\}\\ \ldots\end{array}$$

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

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

Example The append-Function

Consider the concatenation of two lists. In Ocaml, we would write:

$$\begin{array}{l}\textbf{let rec}\ \mathsf{app} = \textbf{fun}\ x \to \textbf{match}\ x\ \textbf{with}\\ \qquad\qquad [\,] \to \textbf{fun}\ y \to y\\ \qquad\qquad \mid h :: t \to \textbf{fun}\ y \to h :: \mathsf{app}\ t\ y\\ \textbf{in}\ \mathsf{app}\ [1;2]\ [3]\end{array}$$

The analysis then results in:

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

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

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.,

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

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

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

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:

$c\,1$

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

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

Function Application:

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

Case Distinction 2:

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

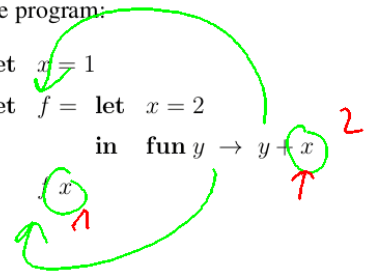
if $p_i \equiv c\, z_1 \ldots z_k$ is the first pattern which matches $c\, v_1 \ldots v_k$:-)

4.4 Application: Inlining

Problem:

- global variables. The program:

$$\begin{array}{l} \mathbf{let}\ \ x = 1 \\ \mathbf{in\ let}\ \ f = \ \mathbf{let}\ \ x = 2 \\ \qquad\qquad\qquad\ \ \mathbf{in}\ \ \mathbf{fun}\ y \;\to\; y + x \\ \mathbf{in}\ \ \ \ f\ x \end{array}$$



... computes something else than:

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

- recursive functions. In the definition:

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

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:

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

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:

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

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 :-)

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:

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

•• 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:

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

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

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

```
   let  x = 1
in let  x_1 = 2
in let  f =  let  x_1 = 2
             in   fun y  →  y + x_1
in      f x
```

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


```
   let  x = 1
in let  x_1 = 2
in let  f =  let  x_1 = 2
             in   fun y  →  y + x_1
in      f x
```

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


```
   let  x = 1
in let  x_1 = 2
in let  f = fun y  →  y + x_1
in      let  y = x
        in   y + x_1
```

Removing variable-variable-assignments, we arrive at:


```
   let  x = 1
in let  x_1 = 2
in let  f = fun y  →  y + x_1
in      x + x_1
```

Idea 2:

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

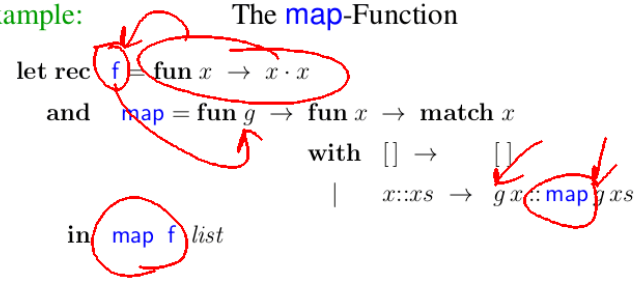
Example: The map-Function

```
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
```

820

- 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
```


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
```

820

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 $x \to x \cdot x$:-)
- 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 actual parameter f in the application map g is always fun $x \to x \cdot x$:-)
- 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
```


Removing useless definitions and variable-variable assignments yields:

```
h = fun x -> match x
             with [] ->      []
                | x::xs ->   x ∗ x :: h xs
```


Removing useless definitions and variable-variable assignments yields:

```
h = fun x -> match x
             with [] ->      []
                | x::xs ->   x ∗ x :: h xs
```


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:

```
map = fun f -> fun l -> match l with [] -> []
                      | x::xs -> f x :: map f xs)
```

```
filter = fun p → fun l → match l with [] → []
                        | x::xs → if p x then x :: filter p xs
                                  else filter p xs)

foldl = fun f → fun a → fun l → match l with [] → a
                                | x::xs → foldl f (f a x) xs)
```



```
id     = fun x → x

comp   = fun f → fun g → fun x → f (g x)

comp1  = fun f → fun g → fun x1 → fun x2 →
           f (g x1) x2

comp2  = fun f → fun g → fun x1 → fun x2 →
           f x1 (g x2)
```


Example:

```
sum    = foldl (+) 0
length = let f = map (fun x → 1)
         in comp sum f
dev    = fun l → let s1   = sum l
                     n    = length l
                     mean = s1/n
                     l1   = map (fun x → x − mean) l
                     l2   = map (fun x → x · x) l1
                     s2   = sum l2
                 in s2/n
```


Observations:

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

  length could also be implemented as:

```
length = let f = fun a → fun x → a + 1
         in foldl f 0
```

- This implementation avoids to create intermediate lists !!!

Example:

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

Observations:

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

  length could also be implemented as:

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

- This implementation avoids to create intermediate lists !!!

Simplification Rules:

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

Simplification Rules:

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

Warning:

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

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

Simplification Rules:

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

Warning:

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

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

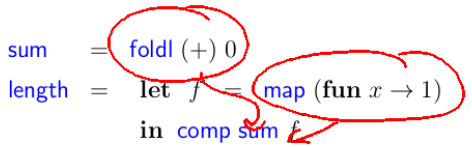
Example, optimized:

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

Example, optimized:

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

Example:



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

Example, optimized:

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

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:

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

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

Extension: Tabulation

If the list has been created by tabulation of a function, the creation of the list sometimes can be avoided ...

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

int → (int → 'a) → int → 'a list
(int → 'a) → int → 'a list

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:

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

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:

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

Extension (2): List Reversals

Sometimes, the ordering of lists or arguments is reversed:

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

Extension (2): List Reversals

Sometimes, the ordering of lists or arguments is reversed:

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