Script **generated by TTT**

Title: Seidl: Programmoptimierung (27.01.2016)

Date: Wed Jan 27 10:22:47 CET 2016

Duration: 89:32 min

Pages: 66

Accordingly, we have for abs :

$$\begin{aligned}\text{let abs} \;=\; \text{fun}\; x \to \;\; & \text{let}\;\; z = (x, -x) \\ & \text{in max}\; z\end{aligned}$$

4.2 A Simple Value Analysis

Idea

For every subexpression e we collect the set $[e]^\sharp$ of possible values of e ...

Let V denote the set of occurring (classes of) constants, functions as well as applications of constructors and operators. As our lattice, we choose:

$$\mathbb{V} \;=\; 2^V$$

As usual, we put up a constraint system:

- If e is a value, i.e., of the form: $b, c\, e_1 \ldots e_k, (e_1, \ldots, e_k)$, an operator application or $\text{fun}\, x \to e$ we generate the constraint:
$$[e]^\sharp \supseteq \{e\}$$
- If $e \equiv (e_1\; e_2)$ and $f \equiv \text{fun}\; x \to e'$, then
$$\begin{array}{lll}[\![e]\!]^\sharp & \supseteq & (f \in [\![e_1]\!]^\sharp)\,?\,[\![e']\!]^\sharp \;:\; \emptyset \\ [\![x]\!]^\sharp & \supseteq & (f \in [\![e_1]\!]^\sharp)\,?\,[\![e_2]\!]^\sharp \;:\; \emptyset\end{array}$$

 ...

- If $e \equiv \text{let}\; x_1 = e_1 \;\text{in}\; e_0$, then we generate:
$$\begin{array}{lll}[\![x_1]\!]^\sharp & \supseteq & [\![e_1]\!]^\sharp \\ [\![e]\!]^\sharp & \supseteq & [\![e_0]\!]^\sharp\end{array}$$
- Analogously for $t \equiv \text{letrec}\; x_1 = e_1 \ldots x_k = e_k \;\text{in}\; e_0$:
$$\begin{array}{lll}[\![x_i]\!]^\sharp & \supseteq & [\![e_i]\!]^\sharp \\ [\![t]\!]^\sharp & \supseteq & [\![e_0]\!]^\sharp\end{array}$$

- 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 $e \equiv \textbf{match}\ e_0\ \textbf{with}\ p_1\ \to\ e_1\ |\ \ldots\ |\ p_k\ \to\ e_k$.
  Then we generate for $p_i \equiv b$ (basic value),

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

...

- 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 $e \equiv \textbf{match}\ e_0\ \textbf{with}\ p_1\ \to\ e_1\ |\ \ldots\ |\ p_k\ \to\ e_k$.
  Then we generate for $p_i \equiv b$ (basic value),

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

...

793

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

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

Let V denote the set of occurring (classes of) constants, functions as well as applications of constructors and operators. As our lattice, we choose:

$$\mathbb{V} = 2^V$$

As usual, we put up a constraint system:

- If e is a value, i.e., of the form: $b, c\, e_1 \ldots e_k, (e_1, \ldots, e_k)$, an operator application or $\textbf{fun}\, x\ \to\ e$ we generate the constraint:

$$[\![e]\!]^\sharp \supseteq \{e\}$$

- If $e \equiv (e_1\ e_2)$ and $f \equiv \textbf{fun}\ x\ \to\ e'$, then

$$\begin{array}{lll} [\![e]\!]^\sharp & \supseteq & (f \in [\![e_1]\!]^\sharp)\,?\,[\![e']\!]^\sharp\ :\ \emptyset \\ [\![x]\!]^\sharp & \supseteq & (f \in [\![e_1]\!]^\sharp)\,?\,[\![e_2]\!]^\sharp\ :\ \emptyset \end{array}$$

...

$e_0 = (x_1, 1::x_2)$

$e_1 = (\text{fun } x \to e')\, e_0$

$[\![e_0]\!]^\#, [\![e_1]\!]^\# \subseteq V$

subexp. of the prog

- 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 $e \equiv \textbf{match}\ e_0\ \textbf{with}\ p_1 \to e_1 \mid \ldots \mid p_k \to e_k$.
  Then we generate for $p_i \equiv b$ (basic value),

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

...

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:

```
let rec app = fun x -> match x with
                         []   -> fun y -> y
                       | h::t -> fun y -> h :: app t y
in app [1; 2] [3]
```

The analysis then results in:

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

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

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 $A = [\![\textsf{app}\;t\;y]\!]^\sharp$:

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

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

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

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

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

796

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

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

796

... in the Example:

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

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

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

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

real values: [[1;2];3], [2;3], [3]

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

$$
\begin{array}{l}
c\, 1 \\
[1; 2] = ::\ 1\ (::\ 2\ [\,]) \\
(\mathbf{fun}\, x \to x{::}y, \{y \mapsto [5]\})
\end{array}
$$

4.4 Application: Inlining

Problem

- global variables. The program:

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

... computes something else than:

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

- recursive functions. In the definition:

$$\text{foo} = \mathbf{fun}\, y \to \text{foo}\, y$$

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.

... computes something else than:

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

- recursive functions. In the definition:

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

foo should better not be substituted.

... computes something else than:

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

- recursive functions. In the definition:

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

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.

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:

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

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

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

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

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

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

... in the Example:

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

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

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

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

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

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

$\Longrightarrow$ now we can apply inlining :

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

Removing variable-variable-assignments, we arrive at:

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

Idea 2

→ We apply our value analysis.

→ We ignore global variables.

→ We only substitute functions without free variables.

Example: The map-Function

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

Idea 2

→ We apply our value analysis.

→ We ignore global variables.

→ We only substitute functions without free variables.

Example: The map-Function

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

817

Idea 2

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

Example: The map-Function

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

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

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

The inner occurrence of map g can be replaced with h

⟹ fold-Transformation.

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

Inlining the function g yields:

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

Removing useless definitions and variable-variable assignments yields:

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

Idea 2

→ We apply our value analysis.

→ We ignore global variables.

→ We only substitute functions without free variables.

Example: The map-Function

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

- 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:

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

Removing useless definitions and variable-variable assignments yields:

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

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:

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

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

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

$$
\begin{array}{lll}
\mathsf{id} & = & \textbf{fun}\ x \to x \\
\\
\mathsf{comp} & = & \textbf{fun}\ f \to \textbf{fun}\ g \to \textbf{fun}\ x \to f\ (g\ x) \\
\\
\mathsf{comp}_1 & = & \textbf{fun}\ f \to \textbf{fun}\ g \to \textbf{fun}\ x_1 \to \textbf{fun}\ x_2 \to \\
& & \quad f\ (g\ x_1)\ x_2 \\
\\
\mathsf{comp}_2 & = & \textbf{fun}\ f \to \textbf{fun}\ g \to \textbf{fun}\ x_1 \to \textbf{fun}\ x_2 \to \\
& & \quad f\ x_1\ (g\ x_2)
\end{array}
$$

Example

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

Example

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

Observations

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

length could also be implemented as:

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

- This implementation avoids to create intermediate lists !!!

Example

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

825

Observations

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

length could also be implemented as:

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

- This implementation avoids to create intermediate lists !!!

826

Example

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

Observations

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

  length could also be implemented as:

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

- This implementation avoids to create intermediate lists !!!

Simplification Rules

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

Simplification Rules

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

Simplification Rules

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

Simplification Rules

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

Caveat

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

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

Simplification Rules

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

foldl ~~[illegible]~~ ~~[illegible]~~ ~~[illegible]~~ h [illegible]

Caveat

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

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

Simplification Rules

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

Caveat

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

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

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:

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

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

Example, optimized:

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

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:

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

- 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:

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

- 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 ...

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

$f \quad n$

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

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

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

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

833