Script generated by TTT

Title: Seidl: Programmoptimierung (21.01.2013)

Date: Mon Jan 21 14:03:33 CET 2013

Duration: 89:07 min

Pages: 45

4.1 A Simple Functional Language

For simplicity, we consider:

$$
\begin{array}{lll}
e & ::= & b \mid (e_1, \ldots, e_k) \mid c\, e_1 \,\ldots\, e_k \mid \mathbf{fun}\, x \to e \\
 & & \mid (e_1\, e_2) \mid (\Box_1\, e) \mid (e_1\, \Box_2\, e_2) \mid \\
 & & \mathbf{let}\; x_1 = e_1 \;\mathbf{in}\; e_0 \mid \\
 & & \mathbf{match}\; e_0 \;\mathbf{with}\; p_1 \to e_1 \;\mid \ldots \mid\; p_k \to e_k \\
p & ::= & b \mid x \mid c\, x_1 \ldots x_k \mid (x_1, \ldots, x_k) \\
t & ::= & \mathbf{let\ rec}\; x_1 = e_1 \;\mathbf{and} \ldots \mathbf{and}\; x_k = e_k \;\mathbf{in}\; e
\end{array}
$$

where b is a constant, x is a variable, c is a (data-)constructor and $\Box_i$ are i-ary operators.

(1 2)

4.1 A Simple Functional Language

For simplicity, we consider:

$$
\begin{array}{lll}
e & ::= & b \mid (e_1, \ldots, e_k) \mid c\, e_1 \,\ldots\, e_k \mid \mathbf{fun}\, x \to e \\
 & & \mid (e_1\, e_2) \mid (\Box_1\, e) \mid (e_1\, \Box_2\, e_2) \mid \\
 & & \mathbf{let}\; x_1 = e_1 \;\mathbf{in}\; e_0 \mid \\
 & & \mathbf{match}\; e_0 \;\mathbf{with}\; p_1 \to e_1 \;\mid \ldots \mid\; p_k \to e_k \\
p & ::= & b \mid x \mid c\, x_1 \ldots x_k \mid (x_1, \ldots, x_k) \\
t & ::= & \mathbf{let\ rec}\; x_1 = e_1 \;\mathbf{and} \ldots \mathbf{and}\; x_k = e_k \;\mathbf{in}\; e
\end{array}
$$

where b is a constant, x is a variable, c is a (data-)constructor and $\Box_i$ are i-ary operators.

785

4.1 A Simple Functional Language

For simplicity, we consider:

$$
\begin{array}{lll}
e & ::= & b \mid (e_1, \ldots, e_k) \mid c\, e_1 \,\ldots\, e_k \mid \mathbf{fun}\, x \to e \\
 & & \mid (e_1\, e_2) \mid (\Box_1\, e) \mid (e_1\, \Box_2\, e_2) \mid \\
 & & \mathbf{let}\; x_1 = e_1 \;\mathbf{in}\; e_0 \mid \\
 & & \mathbf{match}\; e_0 \;\mathbf{with}\; p_1 \to e_1 \;\mid \ldots \mid\; p_k \to e_k \\
p & ::= & b \mid x \mid c\, x_1 \ldots x_k \mid (x_1, \ldots, x_k) \\
t & ::= & \mathbf{let\ rec}\; x_1 = e_1 \;\mathbf{and} \ldots \mathbf{and}\; x_k = e_k \;\mathbf{in}\; e
\end{array}
$$

where b is a constant, x is a variable, c is a (data-)constructor and $\Box_i$ are i-ary operators.

785

... in the Example:

A definition of max may look as follows:

$$
\begin{array}{lll}
\textbf{let } \mathsf{max} \; = \; \textbf{fun } x \rightarrow & \textbf{match } x \textbf{ with } (x_1, x_2) \; \rightarrow \; (& \\
& \quad \textbf{match } x_1 < x_2 & \\
& \quad \textbf{with} \;\; \mathsf{True} \; \rightarrow \; x_2 & \\
& \quad\;\;\; | \;\;\; \mathsf{False} \; \rightarrow \; x_1 & \\
& \quad) &
\end{array}
$$

in e

Accordingly, we have for abs :

$$
\begin{array}{ll}
\textbf{let } \mathsf{abs} \; = \; \textbf{fun } x \rightarrow & \textbf{let } z = (x, -x) \\
& \textbf{in } \mathsf{max} \; z
\end{array}
$$

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 $\textbf{fun } x \rightarrow e$ we generate the constraint:

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

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

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

...

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 $\textbf{fun } x \rightarrow e$ we generate the constraint:

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

- If $e \equiv (e_1\; e_2)$ and $f \equiv \textbf{fun } x \; \rightarrow \; e'$, 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 $e \equiv \textbf{let}\ x_1 = e_1\ \textbf{in}\ e_0$ then we generate:

$$[\![x_1]\!]^\sharp \supseteq [\![e_1]\!]^\sharp$$
$$[\![e]\!]^\sharp \supseteq [\![e_0]\!]^\sharp$$

- Analogously for $t \equiv \textbf{letrec}\ x_1 = e_1 \ldots x_k = e_k\ \textbf{in}\ e_0$:

$$[\![x_i]\!]^\sharp \supseteq [\![e_i]\!]^\sharp$$
$$[\![t]\!]^\sharp \supseteq [\![e_0]\!]^\sharp$$

- 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 $e \equiv \textbf{match}\ e_0\ \textbf{with}\ p_1 \to e_1 \mid \ldots \mid p_k \to e_k$.
  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

$$[\![e]\!]^\sharp \supseteq (v \in [\![e_0]\!]^\sharp)\,?\,[\![e_i]\!]^\sharp : \emptyset$$
$$[\![y_j]\!]^\sharp \supseteq (v \in [\![e_0]\!]^\sharp)\,?\,[\![e'_j]\!]^\sharp : \emptyset$$

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

$$[\![e]\!]^\sharp \supseteq (v \in [\![e_0]\!]^\sharp)\,?\,[\![e_i]\!]^\sharp : \emptyset$$
$$[\![y_j]\!]^\sharp \supseteq (v \in [\![e_0]\!]^\sharp)\,?\,[\![e'_j]\!]^\sharp : \emptyset$$

If $p_i \equiv y$, then

$$[\![e]\!]^\sharp \supseteq [\![e_i]\!]^\sharp$$
$$[\![y]\!]^\sharp \supseteq [\![e_0]\!]^\sharp$$

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

$$[\![e]\!]^\sharp \supseteq (v \in [\![e_0]\!]^\sharp)\,?\,[\![e_i]\!]^\sharp : \emptyset$$
$$[\![y_j]\!]^\sharp \supseteq (v \in [\![e_0]\!]^\sharp)\,?\,[\![e'_j]\!]^\sharp : \emptyset$$

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

$$[\![e]\!]^\sharp \supseteq (v \in [\![e_0]\!]^\sharp)\,?\,[\![e_i]\!]^\sharp : \emptyset$$
$$[\![y_j]\!]^\sharp \supseteq (v \in [\![e_0]\!]^\sharp)\,?\,[\![e'_j]\!]^\sharp : \emptyset$$

If $p_i \equiv y$, then

$$[\![e]\!]^\sharp \supseteq [\![e_i]\!]^\sharp$$
$$[\![y]\!]^\sharp \supseteq [\![e_0]\!]^\sharp$$

Example The append-Function

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

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

The analysis then results in:

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

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

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

$[e]^\# \ni c_0\, e_1 \cdots e_2$

$[e_1]^\# \ni c_1 \boxed{\ }$ $\quad [e_2]^\# \ni c_2 \boxed{\ }$

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

$c(c_1 \Box, c_2 \Box, \ldots)$

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

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

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

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

Example The append-Function

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

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

The analysis then results in:

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

... in the Example:

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

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

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

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

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:

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

Examples for Values:

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

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

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:

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

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

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

Operator applications are treated analogously!

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!

$(e_1, \eta) \Rightarrow v_1 \quad (e_2, \eta) \Rightarrow v_2 \quad v_1 + v_2 = 5$

$(e_1 + e_2, \eta) \Rightarrow (5$

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

Global Definition:

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

Function Application:

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

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 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:

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

Variables:

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

Correctness of the Analysis:

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

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

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

Conclusion:

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

4.4 Application: Inlining

Problem:

- global variables. The program:

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

4.4 Application: Inlining

Problem:

- global variables. The program:

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

... computes something else than:

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

- recursive functions. In the definition:

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

foo should better not be substituted :-)

... computes something else than:

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

- recursive functions. In the definition:

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

foo should better not be substituted :-)

4.4 Application: Inlining

Problem:

- global variables. The program:

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

... computes something else than:

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

- recursive functions. In the definition:

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

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:

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

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:

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

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 :-)

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 $e \equiv \textbf{match } e_0 \ldots c\, x_1 \ldots x_k \to e_i \ldots,$

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

$$D[e_0] = D$$

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

... in the Example:

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

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

⟹ we first duplicate the definition of x_1 :

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

⟹ 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 :

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

Removing variable-variable-assignments, we arrive at: