... in the Example:

A definition of max may look as follows:

$$
\begin{array}{l}
\textbf{let } \mathsf{max} \;=\; \textbf{fun } x \to \;\; \textbf{match } x \textbf{ with } (x_1, x_2) \;\to\; (\\
\qquad \textbf{match } x_1 < x_2 \\
\qquad\;\; \textbf{with } \;\; \mathsf{True} \;\to\; x_2 \\
\qquad\qquad \mid \;\;\; \mathsf{False} \;\to\; x_1 \\
\qquad)
\end{array}
$$

Accordingly, we have for abs :

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

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 $\mathbf{fun}\, x \;\rightarrow\; e$ we generate the constraint:

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

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

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

...

- 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 ...

- If $e \equiv \mathbf{let}\ x_1 = e_1\ \mathbf{in}\ e_0$, then we generate:

$$\begin{array}{lcl} [\![x_1]\!]^\sharp & \supseteq & [\![e_1]\!]^\sharp \\ [\![e]\!]^\sharp & \supseteq & [\![e_0]\!]^\sharp \end{array}$$

- 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 $p_i \equiv b$,

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

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:

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

The analysis then results in:

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

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

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:

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

Examples for Values:

$$c\, 1$$

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

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

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

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:

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

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

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

Operator applications are treated analogously!

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

Global Definition:

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

Function Application:

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

Case Distinction 1:

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

if $p_i \equiv b$ is the first pattern which matches b :-)

Case Distinction 2:

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

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:

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

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

:-)

Case Distinction 4:

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

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

Local Definitions:

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

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:

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

... computes something else than:

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

- recursive functions. In the definition:

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

foo should better not be substituted :-)

Idea 1:

- → First, we introduce unique variable names.
- → Then, we only substitute functions which are staticly within the scope of the same global variables as the application :-)
- → For every expression, we determine all function definitions with this property :-)

Let $D = D[e]$ denote the set of definitions which staticly arrive at e.

- If $e \equiv \textbf{let}\ x_1 = e_1\ \textbf{in}\ e_0$ then:

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

item 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:

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

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

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

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

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

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

$\Longrightarrow$ now we can apply inlining :

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

Removing variable-variable-assignments, we arrive at:

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