4 Optimization of Functional Programs

Example:

\[
\text{let rec } \text{fac } x = \begin{cases} 
1 & \text{if } x \leq 1 \\
 x \cdot \text{fac} (x - 1) & \text{else}
\end{cases}
\]

- There are no basic blocks
- There are no loops
- Virtually all functions are recursive
Strategies for Optimization:

⇒ Improve specific inefficiencies such as:
  • Pattern matching
  • Lazy evaluation (if supported ;-) 
  • Indirections — Unboxing / Escape Analysis
  • Intermediate data-structures — Deforestation

⇒ Detect and/or generate loops with basic blocks :-)
  • Tail recursion
  • Inlining
  • let-Floating

Then apply general optimization techniques
... e.g., by translation into C ;-)
Warning:

Novel analysis techniques are needed to collect information about functional programs.

Example: Inlining

```plaintext
let max (x, y) = if x > y then x
else y
let abs z = max (z, -z)
```

As result of the optimization we expect ...
let \( \text{max} (x, y) = \) if \( x > y \) then \( x \) else \( y \)

let \( \text{abs} z = \) let \( x = z \) and \( y = -z \) in if \( x > y \) then \( x \) else \( y \) end

Discussion:

For the beginning, \( \text{max} \) is just a name. We must find out which value it takes at run-time

\[ \rightarrow \text{Value Analysis required} !! \]
Nevin Heintze in the Australian team of the Prolog-Programming-Contest, 1998
The complete picture:
4.1 A Simple Functional Language

For simplicity, we consider:

\[
\begin{align*}
e & ::= b \mid (e_1, \ldots, e_k) \mid c \; e_1 \ldots \; e_k \mid \text{fun} \; x \to e \\
& \quad \mid (e_1 \; e_2) \mid (\ Diamond_1 \; e) \mid (e_1 \; \Diamond_2 \; e_2) \\
\text{let} & \; x_1 = e_1 \; \text{and} \ldots \; \text{and} \; x_k = e_k \; \text{in} \; e_0 \\
\text{match} & \; e_0 \; \text{with} \; p_1 \to e_1 \mid \ldots \mid p_k \to e_k \\
p & ::= b \mid x \mid c \; x_1 \ldots \; x_k \mid (x_1, \ldots, x_k) \\
t & ::= \text{let rec} \; x_1 = e_1 \; \text{and} \ldots \; \text{and} \; x_k = e_k \; \text{in} \; e
\end{align*}
\]

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

- `let rec` only occurs on top-level.
- Functions are always unary. Instead, there are explicit tuples.
- `if`-expressions and case distinction in function definitions is reduced to `match`-expressions.
- In case distinctions, we allow just simple patterns.
  \[\Rightarrow\] Complex patterns must be decomposed ...
- `let`-definitions correspond to basic blocks.
- `Type-annotations` at variables, patterns or expressions could provide further useful information
  — which we ignore.
... in the Example:

A definition of \texttt{max} may look as follows:

\begin{verbatim}
let max = fun x -> match x with (x_1, x_2) -> (match x_1 < x_2 with True -> x_2 | False -> x_1)
\end{verbatim}
Accordingly, we have for \( \text{abs} : \)

\[
\text{let abs } = \text{ fun } x \rightarrow \text{ let } z = (x, -x) \\
\text{ in max } z
\]

### 4.2 A Simple Value Analysis

**Idea:**

For every subexpression \( e \) we collect the set \([e]\) 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:

$$V = 2^V$$

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:

  $${}[e]^\# \supseteq \{e\}$$

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

  $${}[e]^\# \supseteq (f \in [e_1]^\#)?[e']^\# : \emptyset$$

  $${}[x]^\# \supseteq (f \in [e_1]^\#)?[e_2]^\# : \emptyset$$

...
• \text{int-values returned by operators are described by the unevaluated expression;}

Operator applications which return Boolean values, e.g., by \{\text{True, False}\} 

• \text{If } e \equiv \text{let } x_1 = e_1 \text{ and } \ldots \text{ and } x_k = e_k \text{ in } e_0, \text{ then we generate:}

\[
[x_i]^# \supset [e_i]^#
\]

\[
[e]^# \supset [e_0]^#
\]
Assume $e \equiv \text{match } e_0 \text{ with } p_1 \rightarrow e_1 \mid \ldots \mid p_k \rightarrow e_k$.

Then we generate for $p_i \equiv b$,

$$[[e]] \supseteq [[e_i]] : \emptyset$$

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

$$[[e]] \supseteq (v \in [[e_0]] \supseteq [[e_i]] : \emptyset$$

$$[[y_j]] \supseteq (v \in [[e_0]] \supseteq [[e'_j]] : \emptyset$$

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

$$[[e]] \supseteq (v \in [[e_0]] \supseteq [[e_i]] : \emptyset$$

$$[[y_j]] \supseteq (v \in [[e_0]] \supseteq [[e'_j]] : \emptyset$$

If $p_i \equiv y$, then

$$[[e]] \supseteq [[e_i]]$$

$$[[y]] \supseteq [[e_0]]$$
Example: The append-Function

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

```ocaml
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:

- `[[app]]# = {fun x -> match ...}`
- `[[x]]# = {[1;2], [2], []}`
- `[[match ...]]# = {fun y -> y, fun y -> h::app ...}`
- `[[y]]# = {[3]}`

...
\[
\begin{align*}
[h]^\# &= \{1, 2\} \\
[t]^\# &= \{[2], []\} \\
[\text{app } t]^\# &= \{\text{fun } y \rightarrow y, \text{fun } y \rightarrow h :: \text{app }\ldots\} \\
[\text{app } 1; 2]^\# &= \{\text{fun } y \rightarrow y, \text{fun } y \rightarrow h :: \text{app }\ldots\} \\
[\text{app } t \ y]^\# &= \{\text{fun } y \rightarrow y, \text{fun } y \rightarrow h :: \text{app }\ldots\} \\
[\text{app } 1; 2 \ [3]]^\# &= \{[3], h :: \text{app }\ldots\}
\end{align*}
\]

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

\[\implies\] regular tree grammar
... in the Example:

We obtain for \( A = \text{[app t y]}^\# \):

\[
A \rightarrow [3] \mid [h]^\# :: A \\
[h]^\# \rightarrow 1 \mid 2
\]

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

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

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

Examples for Values:

\[
\begin{align*}
\text{c 1} \\
[1; 2] &= :: 1 (:: 2 []) \\
(\text{fun } x \rightarrow x::y, \{ y \mapsto [5] \})
\end{align*}
\]
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:

\[
(e, \eta) \Rightarrow v
\]

Values:

\[
(b, \eta) \Rightarrow b
\]

\[
(\texttt{fun } x \rightarrow e, \eta) \Rightarrow (\texttt{fun } x \rightarrow e, \eta)
\]
\[(e_1, \eta) \Rightarrow v_1 \ldots (e_k, \eta) \Rightarrow v_k\]

\[
\begin{array}{c}
(e_1, \eta) \Rightarrow v_1 \ldots (e_k, \eta) \Rightarrow v_k \\
\hline
(c \, e_1 \ldots e_k, \eta) \Rightarrow c \, v_1 \ldots v_k
\end{array}
\]
\[(e_1, \eta) \Rightarrow v_1 \quad \ldots \quad (e_k, \eta) \Rightarrow v_k\]

\[
\begin{align*}
((e_1, \ldots, e_k), \eta) &\Rightarrow (v_1, \ldots, v_k) \\

\end{align*}
\]

**Global Definition:**

```
let rec ... x = e ... in ...
```

\[
\begin{align*}
(e, \emptyset) &\Rightarrow v \\

\end{align*}
\]

\[
\begin{align*}
(x, \eta) &\Rightarrow v \\

\end{align*}
\]
Function Application:

\[
\begin{align*}
(e_1, \eta) &\Rightarrow (\text{fun } x \rightarrow e, \eta_1) \\
(e_2, \eta) &\Rightarrow v_2 \\
(e, \eta_1 \oplus \{x \mapsto v_2\}) &\Rightarrow v_3 \\
\hline
(e_1 e_2, \eta) &\Rightarrow v_3
\end{align*}
\]