3.4 Wrap-Up

We have considered various optimizations for improving hardware utilization.

Arrangement of the Optimizations:

- First, global restructuring of procedures/functions and of loops for better memory behavior 😊
- Then local restructuring for better utilization of the instruction set and the processor parallelism 😊
- Then register allocation and finally,
- Peephole optimization for the final kick ...

4 Optimization of Functional Programs

Example:

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

- There are no basic blocks 😦
- There are no loops :-(
- Virtually all functions are recursive :-(

<table>
<thead>
<tr>
<th>Procedures</th>
<th>Tail Recursion + Inlining</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Stack Allocation</td>
</tr>
<tr>
<td>Loops</td>
<td>Iteration Recordering</td>
</tr>
<tr>
<td></td>
<td>→ if-Distribution</td>
</tr>
<tr>
<td></td>
<td>→ for-Distribution</td>
</tr>
<tr>
<td></td>
<td>Value Caching</td>
</tr>
<tr>
<td>Bodies</td>
<td>Life-Range Splitting (SSA)</td>
</tr>
<tr>
<td></td>
<td>Instruction Selection</td>
</tr>
<tr>
<td></td>
<td>Instruction Scheduling with</td>
</tr>
<tr>
<td></td>
<td>→ Loop Unrolling</td>
</tr>
<tr>
<td></td>
<td>→ Loop Fusion</td>
</tr>
<tr>
<td>Instructions</td>
<td>Register Allocation</td>
</tr>
<tr>
<td></td>
<td>Peephole Optimization</td>
</tr>
</tbody>
</table>
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

\[ f(x) = u(x) \times x(x) \]
\[ a \rightarrow \]
\[ 1 \]
\[ b \rightarrow \]
\[ 1 \]
\[ c \rightarrow \]
\[ 1 \]
\[ A \rightarrow \]
Warning:

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

Example: Inlining

\[
\text{let } \max(x, y) = \begin{cases} x & \text{if } x > y \\ y & \text{else} \end{cases} \\
\text{let } \text{abs } z = \max(z, -z) 
\]

As result of the optimization we expect ...

Discussion:

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

\[
\implies \text{Value Analysis required}!!
\]

Warning:

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

Example: Inlining

\[
\text{let } \max(x, y) = \begin{cases} x & \text{if } x > y \\ y & \text{else} \end{cases} \\
\text{let } \text{abs } z = \begin{cases} \text{let } x = z \\ \text{in let } y = -z \\ \text{in } \begin{cases} x & \text{if } x > y \\ y & \text{else} \end{cases} \end{cases}
\]

Discussion:

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

\[
\implies \text{Value Analysis required}!!
\]
Warning:

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

Example: Inlining

\[
\text{let } \max (x, y) = \begin{cases} 
  x & \text{if } x > y \\
  y & \text{else}
\end{cases}
\]

\[
\text{let } \text{abs } z = \max (z, -z)
\]

As result of the optimization we expect ...

The complete picture:

4.1 A Simple Functional Language

For simplicity, we consider:

\[
e ::= \begin{cases} 
  b \mid (e_1, \ldots, e_k) \mid c e_1 \ldots e_k \mid \text{fun } x \to c \\
  (e_1, e_2) \mid (\square_1 e) \mid (e_1 \square_2 e_2) \\
  \text{let } x \gets e_0 \text{ in } e_0 \\
  \text{match } e_0 \text{ with } p_1 \to e_1 \mid \ldots \mid p_k \to e_k
\end{cases}
\]

\[
p ::= \begin{cases} 
  b \mid x \mid c x_1 \ldots x_k \mid (x_1, \ldots, x_k)
\end{cases}
\]

\[
t ::= \begin{cases} 
  \text{let rec } x_1 = e_1 \text{ and } \ldots \text{ and } x_k = e_k \text{ in } e
\end{cases}
\]

where \( b \) is a constant, \( x \) is a variable, \( c \) is a (data-)constructor and \( \square_i \) are \( i \)-ary operators.
4.1 A Simple Functional Language

For simplicity, we consider:

\[
\text{max} = \text{fun } \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}
\]

\[
\begin{array}{l}
\begin{align*}
& \text{let } \mathbf{rec} \ e_1 = e_2 \text{ in } e_3 \mid e_1 = e_2 \\
& \text{match } \mathbf{rec} \ e_0 \text{ with } p_1 \rightarrow e_1 \mid \ldots \\
& \quad \text{let } x_k = e_k \text{ in } e_k \\
& \quad \text{match } e_0 \text{ with } p_1 ightarrow e_1 \mid \ldots \\
& \quad p \ ::= \ b \mid \text{let } \mathbf{rec} \ x_1 = e_1 \text{ and } \ldots \text{and } x_k = e_k \text{ in } e \}
\end{align*}
\end{array}
\]

where \( b \) is a constant, \( x \) is a variable, \( c \) is a (data-)constructor and \( \Box_i \) are 1-ary operators.

Discussion:

- \textbf{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.
  \quad Complex patterns must be decomposed …
- \textbf{let}-definitions correspond to basic blocks.
- Type-annotations at variables, patterns or expressions could provide further useful information.
  \quad which we ignore :-)}
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.
  \[
  \Rightarrow \text{Complex patterns must be decomposed} \ldots
  \]
- let-definitions correspond to basic blocks •
- **Type-annotations** at variables, patterns or expressions could provide further useful information
  — which we ignore •

Accordingly, we have for **abs**:

\[
\text{let } \textbf{abs} = \text{fun } x \Rightarrow \text{let } y = (x, -x) \text{ in } x \neg
\]

4.2 A Simple Value Analysis

Idea:

For every subexpression \( e \) we collect the set \( \llbracket e \rrbracket \) of possible values of \( e \)...

... in the Example:

A definition of **max** may look as follows:

\[
\text{let } \textbf{max} = \text{fun } x \Rightarrow \text{match } x \text{ with } (x_1, x_2) \Rightarrow \begin{cases}
  x_1 < x_2 & \text{match } x_1 < x_2 \\
  \text{with } \text{True } \Rightarrow x_2 \\
  \text{False } \Rightarrow x_1
\end{cases}
\]

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

\[ \mathcal{V} = 2^\mathcal{V} \]

As usual, we put up a constraint system:

- If \( e \) is a value, i.e., of the form: \( b, c_1, \ldots, c_k, (c_1, \ldots, c_k), \) an operator application or \( \text{fun } x \Rightarrow e \) we generate the constraints:

  \[ \llbracket e \rrbracket \supseteq \{ e \} \]

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

  \[ \llbracket e \rrbracket \supseteq \llbracket f \rrbracket \llbracket e_1 \rrbracket \llbracket e_2 \rrbracket \llbracket e' \rrbracket \]
Accordingly, we have for \texttt{abs}:

\begin{align*}
\text{let } \texttt{abs} &= \text{fun } x \rightarrow \text{let } z = (x, -x) \\
&\text{in } \text{max } z
\end{align*}

\subsection{A Simple Value Analysis}

\textbf{Idea:}

For every subexpression \( e \) we collect the set \([e]^I\) of possible values of \( e \).

\begin{itemize}
  \item int-values returned by operators are described by the unevaluated expression;
  \item Operator applications might return Boolean values or other basic values. Therefore, we do replace tests for basic values by non-deterministic choice ...
  \item If \( e \equiv \text{let } x_1 = e_1 \text{ in } e_0 \), then we generate:
    \[ [x_1]^I \supseteq [e_1]^I \]
    \[ [e]^I \supseteq [e_0]^I \]
\end{itemize}

\begin{itemize}
  \item Assume \( e \equiv \text{match } e_0 \text{ with } p_1 \rightarrow e_1 | \ldots | p_k \rightarrow e_k \).
    Then we generate for \( p_i \equiv b \),
    \[ [e]^I \supseteq \text{[b]}^I \]
    If \( p_i \equiv e y_1 \ldots y_k \) and \( v \equiv e_1' \ldots e_k' \) is a value, then
    \[ [e]^I \supseteq (v \in [e_0]^I) \land [e_1'^I] : \emptyset \]
    \[ [y_j]^I \supseteq (v \in [e_0]^I) \land [e_j'^I] : \emptyset \]
    If \( p_i \equiv (y_1, \ldots, y_k) \) and \( v \equiv (e_1', \ldots, e_k') \) is a value, then
    \[ [e]^I \supseteq (v \in [e_0]^I) \land [e_1'^I] : \emptyset \]
    \[ [y_j]^I \supseteq (v \in [e_0]^I) \land [e_j'^I] : \emptyset \]
    If \( p_i \equiv y \), then
    \[ [e]^I \supseteq [e_0]^I \]
    \[ [y]^I \supseteq [v_0]^I \]
\end{itemize}

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:

\begin{itemize}
  \item If \( e \) is a value, i.e., of the form: \( b, c \cdot e_1 \ldots e_k, (e_1, \ldots, e_k) \), an operator application or \( \text{fun } x \rightarrow e \) we generate the constraint:
    \[ [e]^I \supseteq \{e\} \]
  \item If \( e \equiv (e_1, e_2) \) and \( f \equiv \text{fun } x \rightarrow c \) then
    \[ [e_1]^I \supseteq (f \in [c]^I) \setminus [e_2]^I \land [e_2]^I \]
    \[ [e_2]^I \supseteq (f \in [c]^I) \setminus [e_1]^I \land [e_1]^I \]
    ... \\
\end{itemize}
Example

The append Function

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

\[ \text{let } \text{append} = \text{fun } z \rightarrow \text{match } z \text{ with} \]

\[ \text{app} \text{ g y} \rightarrow \text{fun } g + y \rightarrow \text{app } t \]

The analysis then results in:

\[ \text{app} \text{ g y} = \{ \text{fun } g + y \rightarrow \text{app } t \} \]

The values returned by operators are described by an unevaluated expression.

* Operator applications might return Boolean values or other basic values. Therefore, we do replace tests for basic values by non-deterministic choice...
Example  The append-Function

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

\[
\text{let rec app } = \text{ fun } x \rightarrow \text{ match } x \text{ with } \\
\text{ [ ] } \rightarrow \text{ fun } y \rightarrow y \\
\text{ | } h :: t \rightarrow \text{ fun } y \rightarrow h :: \text{ app } t y \\
\text{ in app [1;2] [3]} \\
\]

The analysis then results in:

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

Values \(c_1 \ldots c_k, (c_1, \ldots, c_k)\) or operator applications \(c_1 \square c_2\) now are interpreted as recursive calls \(c [c_1]^2 \ldots [c_k]^2, ([c_1]^2, \ldots, [c_k]^2]\) or \([c_1]^2 \square [c_2]^2\), respectively.

\(\Rightarrow\)  regular tree grammar
... in the Example:

We obtain for $A = [\text{app } t \ y]^2$:

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

Let $L(c)$ denote the set of terms derivable from $[c]^2$ w.r.t. the regular tree grammar. Thus, e.g.,

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

4.3 An Operadic Semantics

Idea:

We construct a Big-Step operational semantics which evaluates expressions w.r.t. an environment $\cdot$:

Values are of the form:

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

Examples for Values:

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

$$(e, \eta) \Rightarrow v$$

**Values:**

$$(b, \eta) \Rightarrow b$$

$$(\text{fun } x \rightarrow c, \eta) \Rightarrow (\text{fun } x \rightarrow c, \eta)$$

$$(e_1, \eta) \Rightarrow v_1 \ldots (e_k, \eta) \Rightarrow v_k$$

$$(c, e_1 \ldots e_k, \eta) \Rightarrow c v_1 \ldots v_k$$

Operator applications are treated analogously!

**Global Definition:**

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

$$\begin{align*}
\text{let rec } \ldots x = e \ldots \text{ in } \ldots \\
(e, \emptyset) & \Rightarrow v \\
(x, \eta) & \Rightarrow v
\end{align*}$$