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 ...
<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 Reordering</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>
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  ;-)

780
Warning:

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

Example: Inlining

```haskell
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 \( \text{max} (x, y) = \) \( \text{if } x > y \text{ then } x \) 
else \( y \) 

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

Discussion:

For the beginning, \textit{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:

\[ e ::= b \mid (e_1, \ldots, e_k) \mid c \ e_1 \ldots e_k \mid \text{fun} \ x \to e \]
\[ \mid (e_1 \ e_2) \mid (\square_1 \ e) \mid (e_1 \ \square_2 \ e_2) \mid \]
\[ \text{let } x_1 = e_1 \ \text{in} \ e_0 \mid \]
\[ \text{match} \ e_0 \ \text{with} \ p_1 \to e_1 \mid \ldots \mid p_k \to e_k \]

\[ p ::= b \mid x \mid c \ x_1 \ldots x_k \mid (x_1, \ldots, x_k) \]

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

where \( b \) is a constant, \( x \) is a variable, \( c \) is a (data-)constructor and \( \square_i \) 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.
  \[\implies\text{Complex patterns must be decomposed} \ldots\]
- **let**-definitions correspond to basic blocks.
- **Type-annotations** at variables, patterns or expressions could provide further useful information
  — which we ignore.
... in the Example:

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

\begin{verbatim}
let max = fun x -> match x with (x1, x2) -> (
  match x1 < x2
  with  True  -> x2
   |     False -> x1
)
\end{verbatim}
Accordingly, we have for \texttt{abs}:

\begin{verbatim}
let abs = fun x -> let z = (x, -x) in max z
\end{verbatim}

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:

1. 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:

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

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

\[
\llbracket e \rrbracket \# \supseteq (f \in \llbracket e_1 \rrbracket \#) \ ? \llbracket e' \rrbracket \# : \emptyset
\]

\[
\llbracket x \rrbracket \# \supseteq (f \in \llbracket e_1 \rrbracket \#) \ ? \llbracket e_2 \rrbracket \# : \emptyset
\]

...
• If \( e \equiv \text{let } x_1 = e_1 \text{ in } e_0 \), then we generate:

\[
\begin{align*}
[x_1]# & \supset [e_1]# \\
[e]# & \supset [e_0]#
\end{align*}
\]

• Analogously for \( t \equiv \text{letrec } x_1 = e_1 \ldots x_k = e_k \text{ in } e_0 \):

\[
\begin{align*}
[x_i]# & \supset [e_i]# \\
[t]# & \supset [e_0]#
\end{align*}
\]
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 $e \equiv \text{match } e_0 \text{ with } p_1 \rightarrow e_1 \mid \ldots \mid p_k \rightarrow e_k$. Then we generate for $p_i \equiv b$ (basic value),

$$[e]^{\#} \supseteq [e_i]^{\#} : \emptyset$$

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

$$\begin{align*}
[e] &\supset (v \in [e_0]) \triangleright [e_i] : \emptyset \\
[y_j] &\supset (v \in [e_0]) \triangleright [e'_j] : \emptyset
\end{align*}$$

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

$$\begin{align*}
[e] &\supset (v \in [e_0]) \triangleright [e_i] : \emptyset \\
[y_j] &\supset (v \in [e_0]) \triangleright [e'_j] : \emptyset
\end{align*}$$

If $p_i \equiv y$, then

$$\begin{align*}
[e] &\supset [e_i] \\
[y] &\supset [e_0]
\end{align*}$$
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:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>([\text{app}]^#)</td>
<td>{\text{fun} x → \text{match} \ldots}</td>
</tr>
<tr>
<td>([x]^#)</td>
<td>{[1; 2], [2], []}</td>
</tr>
<tr>
<td>([\text{match} \ldots]^#)</td>
<td>{\text{fun} y → y, \text{fun} y → h :: app \ldots}</td>
</tr>
<tr>
<td>([y]^#)</td>
<td>{[3]}</td>
</tr>
</tbody>
</table>

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

\[
\begin{align*}
A & \rightarrow [3] \mid [[h]]^\# :: A \\
[[h]]^\# & \rightarrow 1 \mid 2
\end{align*}
\]

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

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

\[
\begin{align*}
\text{c 1} \\
[1; 2] &= :: 1 \quad (:: 2 \quad []) \\
(\text{fun } x \rightarrow x::y, \{y \mapsto [5]\})
\end{align*}
\]
Expressions are evaluated w.r.t. an environment \( \eta : \text{Vars} \rightarrow \text{Values} \).

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:**

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

\[
(\text{fun } x \rightarrow e, \eta) \Rightarrow (\text{fun } x \rightarrow e, \eta)
\]

\[
(e_1, \eta) \Rightarrow v_1 \ldots (e_k, \eta) \Rightarrow v_k
\]

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

Operator applications are treated analogously!