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:

```
let rec fac x = if x ≤ 1 then 1
                else x · fac (x - 1)
```

- There are no basic blocks
- There are no loops
- Virtually all functions are recursive
4 Optimization of Functional Programs

Example:

```haskell
let rec fac x = if x ≤ 1 then 1 else x * fac (x - 1)
```

- 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 :-(

Example:

```haskell
let rec fac x = if x ≤ 1 then 1 else x * fac (x - 1)
```

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

```
match [l1, l2, l3, ...] with
[ (C1, x1), [ ] ]; (C2, x2, x3) ->
```

```
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

```
let max (x, y) = if x > y then x
               else y

let abs z = max (z, -z)
```

As result of the optimization we expect ...

---

Discussion:

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

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

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

Example: Inlining

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

As result of the optimization we expect ...

Discussion:

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

\[ \Longrightarrow \text{Value Analysis required} !! \]
4.1 A Simple Functional Language

For simplicity, we consider:

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

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

Discussion:

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

4.1 A Simple Functional Language

For simplicity, we consider:

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

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

---

... in the Example:

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

\[
\text{let } \text{max} = \text{fun } x \to \text{match } x \text{ with } (x_1, x_2) \to ( \\
  \text{match } x_1 < x_2 \to \\
  \text{with } \text{True} \to x_2 \\
  \text{| } \text{False} \to x_1 \\
  )
\]

---

Nevin Heintze in the Australian team of the Prolog-Programming-Contest, 1998
Accordingly, we have for \( \text{abs} \):

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

### 4.2 A Simple Value Analysis

**Idea:**

For every subexpression \( e \) we collect the set \([e]^2\) of possible values of \( e \).

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

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 \( b, c_1 \ldots c_k, (e_1, \ldots, e_k) \), an operator application or \( \text{fun } x \rightarrow e \) we generate the constraint:

\[
[e]^2 \supseteq [e]
\]

- If \( e \equiv (c_1, \ldots, c_k) \) and \( f \equiv \text{fun } x \rightarrow e', \) then

\[
[e]^2 \supseteq (f \in [c_1]^2)?[e']^2 : \emptyset
\]

\[
[f]^2 \supseteq (f \in [c_1]^2)?[e_2]^2 : \emptyset
\]

\[...
\]

- 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 ...