We obtain:

```plaintext
for (i = 0; i < N; i++)
  for (j = 0; j < M; j++)
    c[i][j] = 0;
for (k = 0; k < K; k++)
  for (j = 0; j < M; j++)
    c[i][j] = c[i][j] + a[i][k] * b[k][j];
```

Discussion:

- Instead of fusing several loops, we now have distributed the loops :-)
- Accordingly, conditionals may be moved out of the loop  
  if-distribution ...

---

Correctness:

- The read entries (here: no) may not be modified in the remaining body of the loop !!!
- The ordering of the write accesses to a memory cell may not be changed  :-)

---

Now, the two iterations can no longer be exchanged  :-(
- The iteration over j, however, can be duplicated ...
Warning:

Instead of using this transformation, the inner loop could also be optimized as follows:

```
for (i = 0; i < N; i++)
    for (j = 0; j < M; j++)
        t = 0;
        for (k = 0; k < K; k++)
            t = t + a[i][k] * b[k][j];
        c[i][j] = t;
```

Idea:

If we find heavily used array elements $a[e_1] \ldots [e_2]$ whose index expressions stay constant within the inner loop, we could instead also provide auxiliary registers :-)

Warning:

The latter optimization prohibits the former and vice versa …

Discussion:

- so far, the optimizations are concerned with iterations over arrays.
- Cache-aware organization of other data-structures is possible, but in general not fully automatic …

Example:  

```
<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Stacks

```
1
2
3
4
```

```
1
```
**Advantage:**

- The implementation is simple :-(
- The operations push / pop require constant time :-(
- The data-structure may grow arbitrarily :-(

**Disadvantage:**

- The individual list objects may be arbitrarily dispersed over the memory :-(

**Improvement:**

- If the array is full, replace it with another of double size !!!
- If the array drops empty to a quarter, halve the array again !!!

- The extra amortized costs are constant :-(
- The implementation is no longer so trivial :-(

**Discussion:**

- The same idea also works for queues :-(
- Other data-structures are attempted to organize blockwise.
  
  **Problem:** how can accesses be organized such that they refer mostly to the same block ???

  - Algorithms for external data

**Alternative:**

- The implementation is also simple :-(
- The operations push / pop still require constant time :-(
- The data are consecutively allocated; stack oscillations are typically small

  ➞ better Cache behavior !!!
2. Stack Allocation instead of Heap Allocation

Problem:

- Programming languages such as Java allocate all data-structures in the heap — even if they are only used within the current method.
- If no reference to these data survives the call, we want to allocate these on the stack.

⇒ Escape Analysis

Example:

```
Our Pointer Language
```
```
x = new();
y = new();
x[4] = y;
z = y;
ret = z;
```

... could be a possible method body.

Accessible from the outside world are memory blocks which:

- are assigned to a global variable such as `ret`;
- are reachable from global variables.

... in the Example:
```
x = new();
y = new();
x[4] = y;
z = y;
ret = z;
```

Accessible from the outside world are memory blocks which:

- are assigned to a global variable such as `ret`;
- are reachable from global variables.

... in the Example:
```
x = new();
y = new();
x[4] = y;
z = y;
ret = z;
```
Accessible from the outside world are memory blocks which:

- are assigned to a global variable such as `ret`; or
- are reachable from global variables.

... in the Example:

\[
\begin{align*}
x &= \text{new}(); \\
y &= \text{new}(); \\
x[A] &= y; \\
z &= y; \\
ret &= \text{[z]};
\end{align*}
\]

We conclude:

- The objects which have been allocated by the first `new()` may never escape.
- They can be allocated on the stack :-(

Warning:

This is only meaningful if only few such objects are allocated during a method call :-(

If a local `new()` occurs within a loop, we still may allocate the objects in the heap :-(

Extension: Procedures

- We require an interprocedural points-to analysis :-(
- We know the whole program, we can, e.g., merge the control-flow graphs of all procedures into one and compute the points-to information for this.
- Warning: If we always use the same global variables \(y_1, y_2, \ldots\) for (the simulation of) parameter passing, the computed information is necessarily imprecise :-(
- If the whole program is not known, we must assume that each reference which is known to a procedure escapes :-(

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:

\[
\text{let rec } \text{fac } x = \text{ if } x \leq 1 \text{ then } 1 \\
\text{ else } x \text{ fac } (x - 1)
\]

- 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 *:

780
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 ...

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

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

Discussion:

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

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

Warning:

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

Example: Inlining

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

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

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

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

For simplicity, we consider:

\[
\begin{align*}
  c & ::= b \mid (c_1, \ldots, c_k) \mid c_{c_1} \ldots c_k \mid \text{fun } x \rightarrow c \\
                     & \mid (c_1, c_2) \mid (\text{if } c) \mid (c_1 \text{ if } c_2) \\
                      & \mid \text{let } x_1 = c \text{ in } c_0 \\
                      & \mid \text{match } e_0 \text{ with } p_1 \rightarrow c_1 \mid \ldots \mid p_k \rightarrow c_k \\
  p & ::= b \mid x \mid e x_1 \ldots x_k \mid (x_1, \ldots, x_k) \\
  t & ::= \text{let } x_1 = e_1 \text{ and } \ldots \text{ and } x_k = e_k \text{ in } e
\end{align*}
\]

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