Example: Matrix-Matrix Multiplication

\[
\begin{align*}
\text{for } (i = 0; i < N; i++) \\
\text{for } (j = 0; j < M; j++) \\
\text{for } (k = 0; k < K; k++) \\
& c[i][j] = c[i][j] + a[i][k] \cdot b[k][j];
\end{align*}
\]

Over \( b[][] \) the iteration is columnwise.

Exchange the two inner loops

\[
\begin{align*}
\text{for } (i = 0; i < N; i++) \\
\text{for } (k = 0; k < K; k++) \\
\text{for } (j = 0; j < M; j++) \\
& c[i][j] = c[i][j] + a[i][k] \cdot b[k][j];
\end{align*}
\]

Is this permitted ???
## Discussion

- Correctness follows as before.
- A similar idea can also be used for the implementation of multiplication for *row compressed* matrices.
- Sometimes, the program must be *massaged* such that the transformation becomes applicable.
- Matrix-matrix multiplication perhaps requires initialization of the result matrix first ...
for \((i = 0; i < N; i++)\)
\[
\text{for } \ (j = 0; j < M; j++) \ 
\{ \\
\text{\hspace{1em}} c[i][j] = 0; \\
\text{\hspace{1em}} \text{for } \ (k = 0; k < K; k++) \\
\text{\hspace{2em}} c[i][j] = c[i][j] + a[i][k] \cdot b[k][j]; \\
\} \\
\]

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

We obtain:

\[
\text{for } \ (i = 0; i < N; i++) \ 
\{ \\
\text{\hspace{1em}} c[i][j] = 0; \\
\text{\hspace{1em}} \text{for } \ (j = 0; j < M; j++) \\
\text{\hspace{2em}} c[i][j] = c[i][j] + a[i][k] \cdot b[k][j]; \\
\} \\
\]

Discussion

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

Caveat

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

\[
\text{for } \ (i = 0; i < N; i++) \ 
\{ \\
\text{\hspace{1em}} t = 0; \\
\text{\hspace{1em}} \text{for } \ (j = 0; j < M; j++) \\
\text{\hspace{2em}} t = t + a[i][k] \cdot b[k][j]; \\
\text{\hspace{2em}} c[i][j] = t; \\
\} \\
\]
Idea

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

Caveat

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: Stacks

![Stack diagram]

Disadvantage

- The data-structure is bounded.

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 !!!

\[ \Rightarrow \text{The extra amortized costs are constant.} \]
\[ \Rightarrow \text{The implementation is no longer so trivial.} \]
Disadvantage

- The data-structure is bounded.

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

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

Idea

Determine points-to information.
Determine if a created object is possibly reachable from the outside ...

Example: Our Pointer Language

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

... could be a possible method body.
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:

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

We conclude:

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

Caveat

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.
- Caveat: 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.

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:

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

<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 Scheduling</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>Instruction Selection</td>
</tr>
<tr>
<td></td>
<td>Peephole Optimization</td>
</tr>
</tbody>
</table>

<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 Scheduling</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>Instruction Selection</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.

Warning:

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

Example:

Inlining

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

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

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

let \textbf{abs} \ z = \begin{cases} z & \text{let } x = z \\ \text{in } \begin{cases} x & \text{if } x > y \text{ then } x \\ y & \text{else} \end{cases} & \text{let } y = -z \\ \text{in } \begin{cases} x & \text{if } x > y \text{ then } x \\ y & \text{else} \end{cases} \end{cases}

Discussion:

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

\Longrightarrow \textbf{Value Analysis} required !!

The complete picture:

Nevin Heintze in the Australian team of the Prolog-Programming-Contest, 1998
4.1 A Simple Functional Language

For simplicity, we consider:

$$ e ::= b | (c_1, \ldots, c_k) | e_1 \ldots e_k | \text{fun } x \rightarrow e \\
| (e_1, e_2) | (\text{if } e \text{ then } c_1 \text{ else } c_2) | \\
| \text{let } x_1 = e_1 \text{ in } e_0 | \\
\text{match } e_0 \text{ with } p_1 \rightarrow e_1 | \ldots | p_k \rightarrow e_k \\
p ::= b | \text{let } x_1 = e_1 | \ldots | x_k = e_k | (x_1, \ldots, x_k) \\
t ::= \text{let rec } x_1 = e_1 \text{ and } \ldots \text{ and } x_k = e_k \text{ in } e $$

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

Discussion:

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

$\implies$ Value Analysis required
4.1 A Simple Functional Language

For simplicity, we consider:

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

\[ p ::= \ b \ | \ x \ | \ c \ x_1 \ldots x_k \ | \ (x_1, \ldots, x_k) \ |
\text{let} \ x_1 = e_1 \ \text{and} \ldots \ \text{and} \ x_k = e_k \ \text{in} \ e \ |
\]

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

... in the Example:

A definition of \( \max \) may look as follows:

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

\[
\text{let} \ x @ y = \ \text{if} \ x @ y \text{then} \ y \ \text{else} \ y \ |
\text{let} \ x @ y = \ \text{if} \ x @ y \text{then} \ y \ \text{else} \ y \ |
\]
4.1 A Simple Functional Language

For simplicity, we consider:

\[ e ::= b \mid (e_1, \ldots, e_k) \mid c \ e_1 \ldots e_k \mid \text{fun} \ x \to e \mid \begin{array}{l}
(c_1 \ c_2) \\
(e_1 \ e_2)
\end{array} \mid (\begin{array}{l}
\exists \ {e_1} \\
\forall \ {e_2}
\end{array}) \mid (\begin{array}{l}
\exists i \ {e_1} \\
\forall i \ {e_2}
\end{array}) \mid \\
\text{let } x_1 = \ell_{e_1} \text{ in } e_1 \\
\text{match } e_0 \text{ with } p_1 \to e_1 \mid \ldots \mid p_k \to e_k \\
\text{let rec } x_1 \equiv e_1 \text{ and } \ldots \text{ and } x_k \equiv e_k \text{ in } e
\]

where \( b \) is a constant, \( x \) is a variable, \( e \) is a (data-)constructor and \( \exists_1 \) are 1-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.
- **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 \( \text{max} \) may look as follows:

\[
\text{let } \text{max} = \text{fun } x \to \text{match } x \text{ with }
\begin{align*}
\text{True} & \to x_2 \\
\text{False} & \to x_1
\end{align*}
\]
Accordingly, we have for \(\text{abs}\):

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

4.2 A Simple Value Analysis

Idea

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