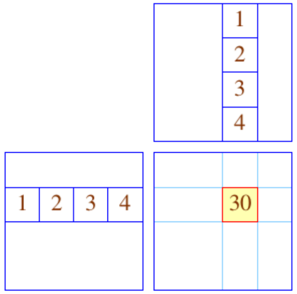
Script **generated by TTT**

Title:	Seidl: Programmoptimierung (21.01.2016)
Date:	Thu Jan 21 08:38:13 CET 2016
Duration:	86:16 min
Pages:	48

Example: Matrix-Matrix Multiplication

```
for $(i = 0; i < N; i{+}{+})$
    for $(j = 0; j < M; j{+}{+})$
        for $(k = 0; k < K; k{+}{+})$
            $c[i][j] = c[i][j] + a[i][k] \cdot b[k][j];$
```

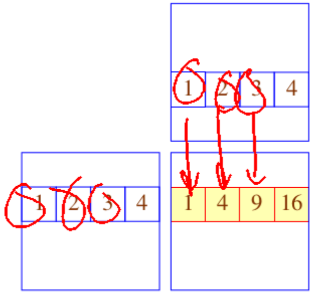
Over $b[][]$ the iteration is columnwise.



Exchange the two inner loops

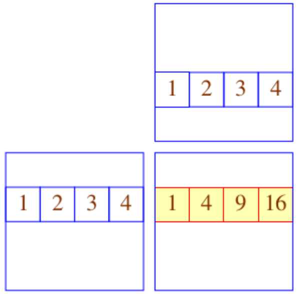
```
for $(i = 0; i < N; i{+}{+})$
    for $(k = 0; k < K; k{+}{+})$
        for $(j = 0; j < M; j{+}{+})$
            $c[i][j] = c[i][j] + a[i][k] \cdot b[k][j];$
```

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 ...



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 $(i = 0; i < N; i++)$
    for $(j = 0; j < M; j++)$ {
        $c[i][j] = 0;$
        for $(k = 0; k < K; k++)$
            $c[i][j] = c[i][j] + a[i][k] \cdot b[k][j];$
    }
```

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

We obtain:

```
for $(i = 0; i < N; i++)$ {
    for $(j = 0; j < M; j++)$ $c[i][j] = 0;$
    for $(k = 0; k < K; k++)$
        for $(j = 0; j < M; j++)$
            $c[i][j] = c[i][j] + a[i][k] \cdot b[k][j];$
}
```

Discussion

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

We obtain:

for $(i = 0; i < N; i++)$ {
    for $(j = 0; j < M; j++)$ $c[i][j] = 0;$
    for $(k = 0; k < K; k++)$
        for $(j = 0; j < M; j++)$
            $c[i][j] = c[i][j] + a[i][k] \cdot b[k][j];$
}

Discussion

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

Caveat

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 $(k = 0; k < K; k++)$
            $t = t + a[i][k] \cdot b[k][j];$
        $c[i][j] = t;$
    }
```

Idea

If we find heavily used array elements $a[e_1] \ldots [e_r]$ 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 ...

Idea

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

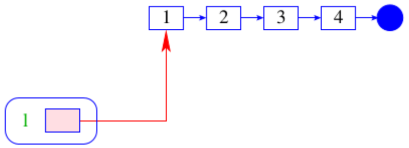
Caveat

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



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.

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 out side ...

Example: Our Pointer Language

$$x = \mathsf{new}();$$
$$y = \mathsf{new}();$$
$$x[A] = y;$$
$$z = y;$$
$$\mathsf{ret} = z;$$

... could be a possible method body.

Accessible from the outside world are memory blocks which:

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

... in the Example:

$$x = \text{new}();$$
$$y = \boxed{\text{new}()};$$
$$x[A] = y;$$
$$z = \boxed{y};$$
$$\text{ret} = \boxed{z};$$

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:

$$x = \text{new}();$$
$$y = \boxed{\text{new}()};$$
$$x[A] = y;$$
$$z = \boxed{y};$$
$$\text{ret} = \boxed{z};$$

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.

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 ...

Procedures:	Tail Recursion + Inlining Stack Allocation
Loops:	Iteration Reordering → if-Distribution → for-Distribution Value Caching
Bodies:	Life-Range Splitting (SSA) Instruction Scheduling with → Loop Unrolling → Loop Fusion
Instructions:	Register Allocation Instruction Selection Peephole Optimization

---

Procedures:	Tail Recursion + Inlining Stack Allocation
Loops:	Iteration Reordering → if-Distribution → for-Distribution Value Caching
Bodies:	Life-Range Splitting (SSA) Instruction Scheduling with → Loop Unrolling → Loop Fusion
Instructions:	Register Allocation Instruction Selection Peephole Optimization

4 Optimization of Functional Programs

Example:

$$\begin{aligned} \text{let rec } \mathsf{fac}\ x \quad = \quad & \text{if } x \leq 1 \text{ then } 1 \\ & \text{else } x \cdot \mathsf{fac}\ (x-1) \end{aligned}$$

- 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.

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

$$\begin{aligned} \text{let } \mathsf{max}\ (x, y) \quad &= \quad \text{if } x > y \text{ then } x \\ & \qquad \text{else } y \\ \text{let } \mathsf{abs}\ z \quad &= \quad \mathsf{max}\ (z, -z) \end{aligned}$$

As result of the optimization we expect ...

$$\begin{array}{lcll} \text{let } \mathsf{max}\,(x,y) & = & \textbf{if } x > y \textbf{ then } x & \\ & & \textbf{else } y & \\ \text{let } \mathsf{abs}\,z & = & \quad\text{let} & x = z \\ & & \text{in let} & y = -z \\ & & \text{in} & \boxed{\begin{array}{l} \textbf{if } x > y \textbf{ then } x \\ \textbf{else } y \end{array}} \end{array}$$

Discussion:

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

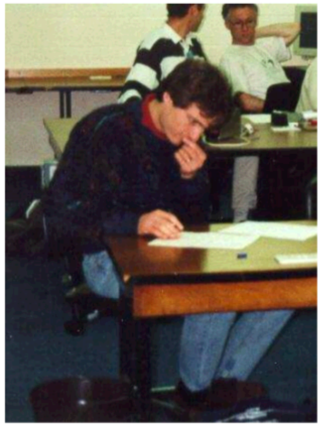
⟹ Value Analysis required !!

$$\begin{array}{lcll} \text{let } \mathsf{max}\,(x,y) & = & \textbf{if } x > y \textbf{ then } x & \\ & & \textbf{else } y & \\ \text{let } \mathsf{abs}\,z & = & \quad\text{let} & x = z \\ & & \text{in let} & y = -z \\ & & \text{in} & \boxed{\begin{array}{l} \textbf{if } x > y \textbf{ then } x \\ \textbf{else } y \end{array}} \end{array}$$

Discussion:

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

⟹ Value Analysis required !!



Nevin Heintze in the Australian team
of the Prolog-Programming-Contest, 1998

The complete picture:

The complete picture:

4.1 A Simple Functional Language

For simplicity, we consider:

$$\begin{array}{lcl} e & ::= & b \mid (e_1, \ldots, e_k) \mid c\, e_1 \ldots e_k \mid \mathbf{fun}\; x \to e \\ & & \mid (e_1\, e_2) \mid (\Box_1\, e) \mid (e_1\, \Box_2\, e_2) \mid \\ & & \mathbf{let}\; x_1 = e_1\; \mathbf{in}\; e_0 \mid \\ & & \mathbf{match}\; e_0\; \mathbf{with}\; p_1 \to e_1 \mid \ldots \mid p_k \to e_k \\ p & ::= & b \mid x \mid c\, x_1 \ldots x_k \mid (x_1, \ldots, x_k) \\ t & ::= & \mathbf{let\; rec}\; x_1 = e_1\; \mathbf{and} \ldots \mathbf{and}\; x_k = e_k\; \mathbf{in}\; e \end{array}$$

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

$$\begin{array}{lcl} \mathbf{let}\; \mathsf{max}\; (x, y) & = & \mathbf{if}\; x > y\; \mathbf{then}\; x \\ & & \mathbf{else}\; y \\ \mathbf{let}\; \mathsf{abs}\; z & = & \quad \mathbf{let}\; x = z \\ & & \mathbf{in\; let}\; y = -z \\ & & \mathbf{in}\; \boxed{\begin{array}{l} \mathbf{if}\; x > y\; \mathbf{then}\; x \\ \mathbf{else}\; y \end{array}} \end{array}$$

Discussion:

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

$\Longrightarrow$ Value Analysis required !!

---

let max x y =

if x > y then x

else y

$: \underset{\uparrow}{'a} \to (\underset{\uparrow}{'a} \to \underset{\uparrow}{\text{bool}})$

4.1 A Simple Functional Language

For simplicity, we consider:

$$\begin{array}{lcl} e & ::= & b \mid (e_1,\ldots,e_k) \mid c\; e_1 \;\ldots\; e_k \mid \mathbf{fun}\; x \to e \\ & & \mid (e_1\; e_2) \mid (\Box_1\; e) \mid (e_1\; \Box_2\; e_2) \mid \\ & & \mathbf{let}\; x_1 = e_1 \;\mathbf{in}\; e_0 \mid \\ & & \mathbf{match}\; e_0 \;\mathbf{with}\; p_1 \to e_1 \mid \ldots \mid p_k \to e_k \\ p & ::= & b \mid x \mid c\; x_1 \ldots x_k \mid (x_1,\ldots,x_k) \\ t & ::= & \mathbf{let\ rec}\; x_1 = e_1 \;\mathbf{and} \ldots \mathbf{and}\; x_k = e_k \;\mathbf{in}\; e \end{array}$$

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

... in the Example:

A definition of max may look as follows:

$$\begin{array}{l} \mathbf{let}\; \mathsf{max} \;=\; \mathbf{fun}\; x \to \quad \mathbf{match}\; x \;\mathbf{with}\; (x_1, x_2) \;\to\; (\\ \qquad\qquad\qquad\qquad \mathbf{match}\; x_1 < x_2 \\ \qquad\qquad\qquad\qquad \;\;\mathbf{with}\;\; \mathsf{True} \;\to\; x_2 \\ \qquad\qquad\qquad\qquad \quad\mid \quad \mathsf{False} \;\to\; x_1 \\ \qquad\qquad\qquad\qquad) \end{array}$$

let max x y =
if x > y then x
else y

$: \; 'a \to ('a \to bool)$

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.

  ⟹ Complex patterns must be decomposed ...
- **let**-definitions correspond to basic blocks.
- Type-annotations at variables, patterns or expressions could provide further useful information

  — which we ignore.

if x > y then x else y

case x > y with true → x | false → y

4.1 A Simple Functional Language

For simplicity, we consider:

$$\begin{array}{lll}
e & ::= & b \mid (e_1, \ldots, e_k) \mid c\, e_1 \ldots e_k \mid \mathbf{fun}\; x \to e \\
 & & \mid (e_1\, e_2) \mid (\Box_1\; e) \mid (e_1 \,\Box_2\, e_2) \mid \\
 & & \mathbf{let}\; x_1 = e_1\; \mathbf{in}\; e_0 \mid \\
 & & \mathbf{match}\; e_0\; \mathbf{with}\; p_1 \to e_1 \;\mid \ldots \mid\; p_k \to e_k \\
p & ::= & b \mid x \mid c\, x_1 \ldots x_k \mid (x_1, \ldots, x_k) \\
t & ::= & \mathbf{let\ rec}\; x_1 = e_1\; \mathbf{and} \ldots \mathbf{and}\; x_k = e_k\; \mathbf{in}\; e
\end{array}$$

where b is a constant, x is a variable, c is a (data-)constructor and $\Box_i$ 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.

  ⟹ Complex patterns must be decomposed ...
- **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 max may look as follows:

$$\begin{array}{l}
\mathbf{let}\ \text{max} = \mathbf{fun}\; x \to \quad \mathbf{match}\; x\; \mathbf{with}\; (x_1, x_2) \to (\\
\qquad\qquad\qquad\qquad \mathbf{match}\; x_1 < x_2 \\
\qquad\qquad\qquad\qquad \mathbf{with}\;\; \text{True} \to x_2 \\
\qquad\qquad\qquad\qquad\quad \mid\;\; \text{False} \to x_1 \\
\qquad\qquad\qquad\quad)
\end{array}$$

Accordingly, we have for abs :

$$\begin{array}{lll} \textbf{let}\ \text{abs} & = \ \textbf{fun}\ x \rightarrow & \textbf{let}\ z = (x, -x) \\ & & \textbf{in}\ \text{max}\ z \end{array}$$

4.2 A Simple Value Analysis

Idea

For every subexpression e we collect the set $[e]^\sharp$ of possible values of e ...