Script generated by TTT

Title: Seidl: Programmoptimierung (29.01.2014)

Date: Wed Jan 29 08:40:13 CET 2014

Duration: 80:47 min

Pages: 27

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 Selection Instruction Scheduling with → Loop Unrolling → Loop Fusion
Instructions:	Register Allocation Peephole Optimization

4 Optimization of Functional Programs

Example:

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

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

4 Optimization of Functional Programs

Example:

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

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

4 Optimization of Functional Programs

Example:

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

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

784

Strategies for Optimization:

$\Longrightarrow$ Improve specific inefficiencies such as:

- Pattern matching
- Lazy evaluation (if supported ;-)
- Indirections — Unboxing / Escape Analysis
- Intermediate data-structures — Deforestation

$\Longrightarrow$ Detect and/or generate loops with basic blocks :-)

- Tail recursion
- Inlining
- **let**-Floating

Then apply general optimization techniques

... e.g., by translation into C ;-)

match e1, e2, e3 with

| ([1;2;x], [ ], ([3], x1::xs)) →

| ...

| ...

| ...

Strategies for Optimization:

$\Longrightarrow$ Improve specific inefficiencies such as:

- Pattern matching
- Lazy evaluation (if supported ;-)
- Indirections — Unboxing / Escape Analysis
- Intermediate data-structures — Deforestation

$\Longrightarrow$ 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:

$\Longrightarrow$ Improve specific inefficiencies such as:

- Pattern matching
- Lazy evaluation (if supported ;-)
- Indirections — Unboxing / Escape Analysis
- Intermediate data-structures — Deforestation

$\Longrightarrow$ Detect and/or generate loops with basic blocks :-)

- Tail recursion
- Inlining
- **let**-Floating

Then apply general optimization techniques

... e.g., by translation into C ;-)

785

Warning:

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

Example: Inlining

$$
\begin{array}{lll}
\textbf{let}\ \mathsf{max}\,(x,y) & = & \textbf{if}\ \ x > y\ \ \textbf{then}\ \ x \\
 & & \textbf{else}\ \ y \\
\textbf{let}\ \mathsf{abs}\ z & = & \mathsf{max}\,(z, -z)
\end{array}
$$

As result of the optimization we expect ...

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

Discussion:

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

$\Longrightarrow$ Value Analysis required !!

Warning:

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

Example: Inlining

$$
\begin{array}{lll}
\mathbf{let}\ \mathsf{max}\,(x,y) & = & \mathbf{if}\ x > y\ \mathbf{then}\ x \\
 & & \mathbf{else}\ y \\
\mathbf{let}\ \mathsf{abs}\ z & = & \mathsf{max}\,(z, -z)
\end{array}
$$

As result of the optimization we expect ...

Warning:

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

Example: Inlining

$$
\begin{array}{lll}
\mathbf{let}\ \mathsf{max}\,(x,y) & = & \mathbf{if}\ x > y\ \mathbf{then}\ x \\
 & & \mathbf{else}\ y \\
\mathbf{let}\ \mathsf{abs}\ z & = & \mathsf{max}\,(z, -z)
\end{array}
$$

As result of the optimization we expect ...

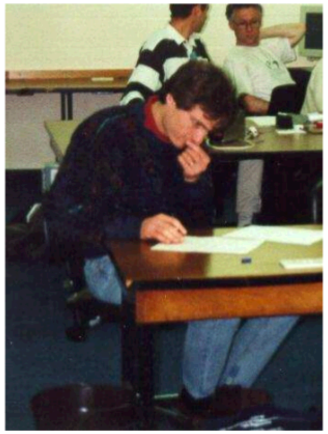
786

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

Discussion:

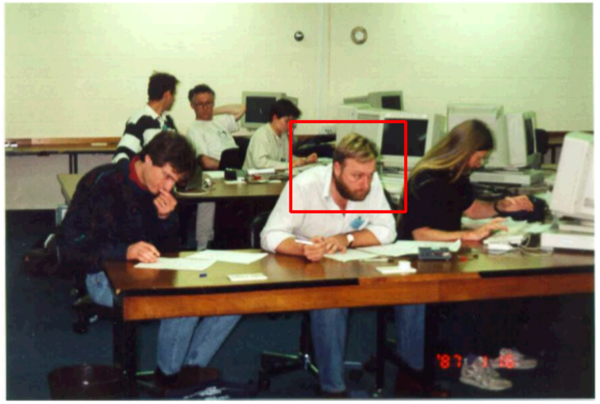
For the beginning, max is just a name. We must find out which value it takes at run-time

$\Longrightarrow$ Value Analysis required !!



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:

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

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.

  $\Longrightarrow$ 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 :-)

4.1 A Simple Functional Language

For simplicity, we consider:

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

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.

  $\Longrightarrow$ 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:

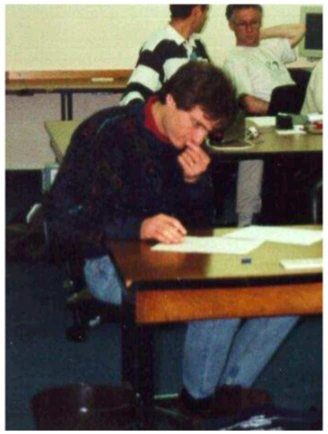
$$
\begin{array}{l}
\textbf{let}\ \mathsf{max}\ =\ \textbf{fun}\ x \rightarrow\ \ \textbf{match}\ x\ \textbf{with}\ (x_1, x_2)\ \rightarrow\ (\\
\qquad\qquad\qquad\qquad \textbf{match}\ x_1 < x_2 \\
\qquad\qquad\qquad\qquad\ \ \textbf{with}\ \ \mathsf{True}\ \rightarrow\ \ x_2 \\
\qquad\qquad\qquad\qquad\quad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

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.

  $\Longrightarrow$ 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 :-)



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

Accordingly, we have for abs :

$$\mathbf{let}\ \mathsf{abs}\ =\ \mathbf{fun}\ x \to\ \ \mathbf{let}\ z = (x, -x)$$
$$\mathbf{in}\ \mathsf{max}\ z$$

4.2 A Simple Value Analysis

Idea:

For every subexpression e we collect the set $[\![e]\!]^\sharp$ 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:

$$\mathbb{V}\ =\ 2^V$$

As usual, we put up a constraint system:

- 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 $\mathbf{fun}\ x \to e$ we generate the constraint:

$$[\![e]\!]^\sharp \supseteq \{e\}$$

- If $e \equiv (e_1\, e_2)$ and $f \equiv \mathbf{fun}\ x \to e'$, then

$$\begin{array}{lll} [\![e]\!]^\sharp & \supseteq & (f \in [\![e_1]\!]^\sharp)\ ?\ [\![e']\!]^\sharp\ :\ \emptyset \\ [\![x]\!]^\sharp & \supseteq & (f \in [\![e_1]\!]^\sharp)\ ?\ [\![e_2]\!]^\sharp\ :\ \emptyset \end{array}$$

...

- If $e \equiv \mathbf{let}\ x_1 = e_1\ \mathbf{in}\ e_0$, then we generate:

$$\begin{array}{lll} [\![x_1]\!]^\sharp & \supseteq & [\![e_1]\!]^\sharp \\ [\![e]\!]^\sharp & \supseteq & [\![e_0]\!]^\sharp \end{array}$$

- Analogously for $t \equiv \mathbf{letrec}\ x_1 = e_1 \ldots x_k = e_k\ \mathbf{in}\ e_0$:

$$\begin{array}{lll} [\![x_i]\!]^\sharp & \supseteq & [\![e_i]\!]^\sharp \\ [\![t]\!]^\sharp & \supseteq & [\![e_0]\!]^\sharp \end{array}$$

- 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 $e \equiv \mathbf{match}\ e_0\ \mathbf{with}\ p_1 \to e_1 \mid \ldots \mid p_k \to e_k$. Then we generate for $p_i \equiv b$ (basic value),

$$[\![e]\!]^\sharp \supseteq [\![e_i]\!]^\sharp \cancel{:\ \emptyset}$$

...