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 :-)

$\Longrightarrow$ 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:

$$x = \mathsf{new}();$$

$$y = \mathsf{new}();$$

$$x[A] = y;$$

$$z = y;$$

$$\mathsf{ret} = \boxed{z};$$

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:

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

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:

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

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:

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

We conclude:

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

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 ...

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:

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

- 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

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

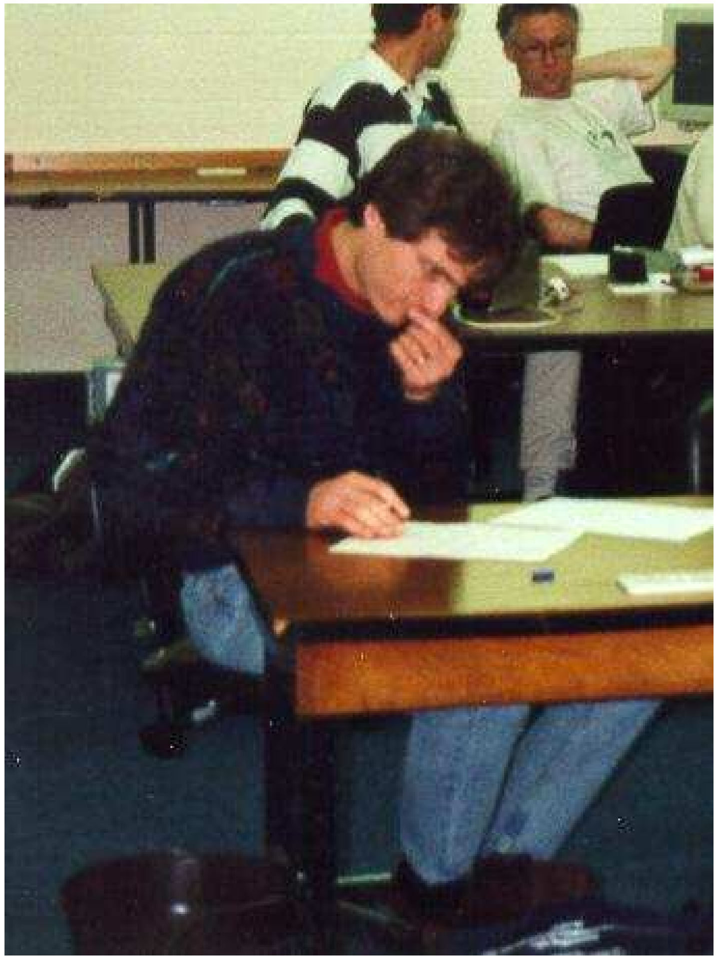
As result of the optimization we expect ...

$$
\begin{array}{llll}
\textbf{let}\ \ \mathsf{max}\ (x,y) & = & \multicolumn{2}{l}{\textbf{if}\ \ x > y\ \ \textbf{then}\ \ x} \\
& & \multicolumn{2}{l}{\textbf{else}\ \ y} \\
\textbf{let}\ \ \mathsf{abs}\ z & = & \quad\textbf{let} & x = z \\
& & \textbf{in let} & y = -z \\
& & \textbf{in} & \boxed{\begin{array}{l}\textbf{if}\ \ x > y\ \ \textbf{then}\ \ x \\ \textbf{else}\ \ y\end{array}}
\end{array}
$$

Discussion:

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

$\Longrightarrow$ 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:

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

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 :-)