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

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

... 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:

```plaintext
x = new();
y = new();
x[A] = 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:

```c
x = new();
y = new();
x[A] = 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:

```plaintext
x = new();
y = new();
x[A] = 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:

```plaintext
x = new();
y = new();
x[A] = y;
z = y;
ret = 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 ;-(

777
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 :-((

778
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 Selection</td>
</tr>
<tr>
<td></td>
<td>Instruction Scheduling with</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>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 ;-)

782
Warning:

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

**Example:**  **Inlining**

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

Discussion:

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

\[ \implies \text{Value Analysis required} !! \]
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:

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

\[
p ::= b | x | c x_1 \ldots x_k | (x_1, \ldots, x_k)
\]

\[
t ::= \text{let rec } x_1 = e_1 \text{ and } \ldots \text{ and } x_k = e_k \text{ in } e
\]

where \( b \) is a constant, \( x \) is a variable, \( c \) is a (data-)constructor and \( \square_i \) 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 :-)

788