For every class $C$ we assume that we are given an address environment $\rho_C$. $\rho_C$ maps every identifier $x$ visible inside $C$ to its decorated relative address $a$. We distinguish:

<table>
<thead>
<tr>
<th>Category</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>global variable</td>
<td>$(G, a)$</td>
</tr>
<tr>
<td>local variable</td>
<td>$(L, a)$</td>
</tr>
<tr>
<td>attribute</td>
<td>$(A, a)$</td>
</tr>
<tr>
<td>virtual function</td>
<td>$(V, b)$</td>
</tr>
<tr>
<td>non-virtual function</td>
<td>$(N, a)$</td>
</tr>
<tr>
<td>static function</td>
<td>$(S, a)$</td>
</tr>
</tbody>
</table>

For virtual functions $x$, we do not store the starting address of the code — but the relative address $b$ of the field of $x$ inside the object :-)

361
For the various of variables, we obtain for the $L$-values:

$$
\text{code}_{L} \quad x \quad \rho = \begin{cases} 
\text{loadr } -3 & \text{if } x = \text{this} \\
\text{loadc } a & \text{if } \rho x = (G, a) \\
\text{loadr } a & \text{if } \rho x = (L, a) \\
\text{loadr } -3 \\
\text{loadc } a \\
\text{add} & \text{if } \rho x = (A, a)
\end{cases}
$$

In particular, the pointer to the current object has relative address -3  :-)

362
Accordingly, we introduce the abbreviated operations:

\[
\text{loadm q} = \text{loaddr} - 3 \\
\phantom{\text{loadm q}} = \text{loadc q} \\
\phantom{\text{loadm q}} = \text{add} \\
\phantom{\text{loadm q}} = \text{load} \\
\text{storem q} = \text{loaddr} - 3 \\
\phantom{\text{storem q}} = \text{loadc q} \\
\phantom{\text{storem q}} = \text{add} \\
\phantom{\text{storem q}} = \text{store}
\]
Discussion:

- Besides storing the current object pointer inside the stack frame, we could have additionally used a specific register $\text{COP}$.
- This register must be updated before calls to non-static member functions and restored after the call.
- We have refrained from doing so since:
  - Only some functions are member functions.
  - We want to reuse as much of the C-machine as possible.
40  Calling Member Functions

Static member functions are considered as ordinary functions :-)  

For non-static member functions, we distinguish two forms of calls:

(1) directly: \( f(e_2, \ldots, e_n) \)
(2) relative to an object: \( e_1.f(e_2, \ldots, e_n) \)

Idea:

- The case (1) is considered as an abbreviation of \( \texttt{this}.f(e_2, \ldots, e_n) \) :-)  
- The object is passed to \( f \) as an implicit first argument :-)
- If \( f \) is non-virtual, proceed as with an ordinary call of a function :-)
- If \( f \) is virtual, insert an indirect call :-)

365
A non-virtual function:

\[
\text{code}_R \ e_1.f \ (e_2, \ldots, e_n) \ \rho = \text{code}_R \ e_n \ \rho \\
\vdots \\
\text{code}_R \ e_2 \ \rho \\
\text{code}_L \ e_1 \ \rho \\
\text{mark} \\
\text{loadc}_-f \\
\text{call} \\
\text{slide} \ m
\]

where \((F, \_f) = \rho_C(f)\)

\(\text{C} = \text{class of } \ e_1\)

\(\text{m} = \text{space for the actual parameters}\)

**Note:**

The pointer to the object is obtained by computing the **L-value** of \( \ e_1 \ :-) \)
A virtual function:

\[
\text{code}_R \ e_1.f \ (e_2, \ldots, e_n) \ \rho = \ \text{code}_R \ e_n \ \rho \\
\ldots \\
\text{code}_R \ e_2 \ \rho \\
\text{code}_L \ e_1 \ \rho \\
\text{mark} \\
\text{loads} \ 2 \\
\text{loadc} \ b \\
\text{add} \ ; \ \text{load} \\
\text{call} \\
\text{slide} \ m
\]

where \((V, b) = \rho_C(f)\)

\(C = \text{class of } e_1\)

\(m = \text{space for the actual parameters}\)
The instruction \textbf{loads} \textit{j} loads relative to the stack pointer:

\[ S[SP+1] = S[SP-j]; \]
\[ SP++; \]
... in the Example:

The recursive call

\[ \text{next} \rightarrow \text{last()} \]

in the body of the virtual method \text{last} is translated into:

\begin{verbatim}
loadm 1
mark
loads 2
loadc 2
add
load
call
\end{verbatim}
41 Defining Member Functions

In general, a definition of a member function for class $C$ looks as follows:

$$d \equiv t \ f \ (t_2 \ x_2, \ldots, t_n \ x_n) \ \{ \ ss \ \}$$

Idea:

- $f$ is treated like an ordinary function with one extra implicit argument
- Inside $f$, a pointer $\text{this}$ to the current object has relative address -3
- Object-local data must be addressed relative to $\text{this}$...
\texttt{\texttt{code}_{D} \ d \ \rho \ = \ _f : \ \texttt{enter \ q} \quad \text{\texttt{// Setting the EP}}

\textbf{alloc m} \quad \text{\texttt{// Allocating the local variables}}

\textbf{code ss \ \rho_1}

\textbf{return} \quad \text{\texttt{// Leaving the function}}

where \quad q \quad = \quad maxS + m \quad \text{\texttt{where}}

\textit{maxS} \quad = \quad \text{maximal depth of the local stack}

\textit{m} \quad = \quad \text{space for the local variables}

\textit{k} \quad = \quad \text{space for the formal parameters (including} \ \textbf{this})

\textit{\rho_1} \quad = \quad \text{local address environment}
... in the Example:

_last:     enter 6       loadm 0     loads 2
alloc 0    storer -3    loadc 2
loadm 1    return       add
loadc 0    load
eq         A: loadm 1   call
jumpz A    mark         storer -3
            return
42 Calling Constructors

Every new object should be initialized by (perhaps implicitly) calling a constructor. We distinguish two forms of object creations:

1. directly: \( x = C(e_2, \ldots, e_n); \)
2. indirectly: \( \text{new } C(e_2, \ldots, e_n) \)

Idea for (2):

- Allocate space for the object and return a pointer to it on the stack;
- Initialize the fields for virtual functions;
- Pass the object pointer as first parameter to a call to the constructor;
- Proceed as with an ordinary call of a (non-virtual) member function \( :) \)
- Unboxed objects are considered later ...
code_{R} \textbf{new} C \left( e_{2}, \ldots, e_{n} \right) \rho = \text{loadc } |C| \newline \text{new} \newline \text{initVirtual } C \newline \text{code}_{R} e_{n} \rho \newline \ldots \newline \text{code}_{R} e_{2} \rho \newline \text{loads } m \quad // \quad \text{loads relative to } SP \quad :-) \newline \text{mark} \newline \text{loadc } _{C} \newline \text{call} \newline \text{pop } m + 1 \newline \newline \text{where} \quad m = \text{space for the actual parameters.} \newline

Before calling the constructor, we initialize all fields of virtual functions. The pointer to the object is copied into the frame by an extra instruction \quad :-)
Assume that the class $C$ lists the virtual functions $f_1, \ldots, f_r$ for $C$ with the offsets and initial addresses: $b_i$ and $a_i$, respectively:

Then:

$$\text{initVirtual } C = \text{ dup}$$
$$\quad \text{loadc } b_1 ; \text{ add}$$
$$\quad \text{loadc } a_1 ; \text{ store}$$
$$\quad \text{pop}$$
$$\quad \ldots$$
$$\quad \text{dup}$$
$$\quad \text{loadc } b_r ; \text{ add}$$
$$\quad \text{loadc } a_r ; \text{ store}$$
$$\quad \text{pop}$$
In general, a definition of a constructor for class $C$ looks as follows:

$$d \equiv C(t_2 x_2, \ldots, t_n x_n) \{ ss \}$$

Idea:

- Treat the constructor as a definition of an ordinary member function :-)

43 Defining Constructors
... in the Example:

```plaintext
_list: enter 3 loada 1 loadc 0
       alloc 0 loadc 1 storem 1
       loadr -4 add      pop
       storem 0 storea 1 return
       pop      pop
```
Discussion:

The constructor may issue further constructors for attributes if desired 

The constructor may call a constructor of the super class $B$ as first action:

\[
\text{code } B(e_2, \ldots, e_n); \rho = \text{code}_R e_n \rho \\
\quad \ldots \\
\quad \text{code}_R e_2 \rho \\
\quad \text{loadr} - 3 \\
\quad \text{mark} \\
\quad \text{loadc}_B \\
\quad \text{call} \\
\quad \text{pop } m + 1
\]

where \( m = \) space for the actual parameters.

The constructor is applied to the current object of the calling constructor!
44 Initializing Unboxed Objects

Problem:
The same constructor application can be used for initializing several variables:

\[ x = x_1 = C(e_2, \ldots, e_n) \]

Idea:

- Allocate sufficient space for a temporary copy of a new C object.
- Initialize the temporary copy.
- Assign this value to the variables to be initialized :-)

379
$$\text{code}_R \ C \ (e_2, \ldots, e_n) \ \rho \ = \ \text{stalloc} \ |C|$$

$$\text{initVirtual} \ C$$

$$\text{code}_R \ e_n \ \rho$$

$$\ldots$$

$$\text{code}_R \ e_2 \ \rho$$

loads $m$

mark

loadc $_C$

call

pop $m + 2$

where $m =$ space for the actual parameters.

**Note:**
The instruction $\text{stalloc} \ m$ is like $\text{malloc} \ m$ but allocates on the stack :-)

We assume that we have assignments between complex types :-)

380
SP = SP+m+1;
S[SP] = SP−m;