Script generated by TTT

Title: Seidl: Virtual Machines (16.06.2014)

Date: Mon Jun 16 10:46:44 CEST 2014

Duration: 43:59 min

Pages: 24

41 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 **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 :-)

A non-virtual function:

$$\mathrm{code_R}\; e_1.f\,(e_2, \ldots, e_n)\; \rho \;=\; \begin{array}{l} \mathrm{code_R}\; e_n\; \rho \\ \ldots \\ \mathrm{code_R}\; e_2\; \rho \\ \mathrm{code_L}\; e_1\; \rho \\ \mathtt{mark} \\ \mathtt{loadc}\; \_f \\ \mathtt{call} \\ \mathtt{slide}\; \mathrm{m} \end{array}$$

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

C = class of e_1

m = space for the actual parameters

Note:

The pointer to the object is obtained by computing the L-value of e_1 :-)

A virtual function:

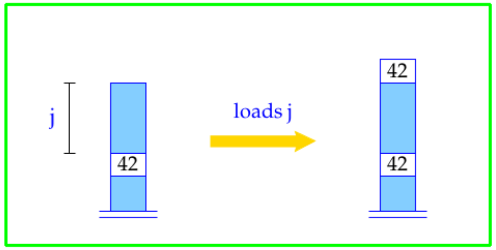
$$\mathrm{code_R}\; e_1.f\,(e_2, \ldots, e_n)\; \rho \;=\; \begin{array}{l} \mathrm{code_R}\; e_n\; \rho \\ \ldots \\ \mathrm{code_R}\; e_2\; \rho \\ \mathrm{code_L}\; e_1\; \rho \\ \mathtt{mark} \\ \mathtt{loads}\; 2 \\ \mathtt{loadc}\; b \\ \mathtt{add};\;\; \mathtt{load} \\ \mathtt{call} \\ \mathtt{slide}\; \mathrm{m} \end{array}$$

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

C = class of e_1

m = space for the actual parameters

The instruction loads j loads relative to the stack pointer:




```
S[SP+1] = S[SP−j];
SP++;
```


... in the Example:

The recursive call

next → last ()

in the body of the virtual method last is translated into:

```
loadm 1
mark
loads 2
loadc 2
add
load
call
```


42 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 **this** to the current object has relative address -3 :-)
- Object-local data must be addressed relative to **this** ...

$$\text{code}_D\ d\ \rho \quad = \quad \_f :$$

```
enter q        // Setting the EP
alloc m        // Allocating the local variables
code ss ρ1
return         // Leaving the function
```

where

q	=	$maxS + m$	where
$maxS$	=	maximal depth of the local stack	
m	=	space for the local variables	
k	=	space for the formal parameters (including **this**)	
ρ_1	=	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
```


A virtual function:

$$
\begin{array}{lcl}
\mathrm{code_R}\ \boxed{e_1.}f\ (e_2, \ldots, e_n)\ \rho & = & \mathrm{code_R}\ e_n\ \rho \\
 & & \ldots \\
 & & \mathrm{code_R}\ e_2\ \rho \\
 & & \boxed{\mathrm{code_L}\ \ e_1\ \ \rho} \\
 & & \mathrm{mark} \\
 & & \mathrm{loads}\ 2 \\
 & & \mathrm{loadc}\ b \\
 & & \mathrm{add}\ ;\ \ \mathrm{load} \\
 & & \mathrm{call} \\
 & & \mathrm{slide}\ m
\end{array}
$$

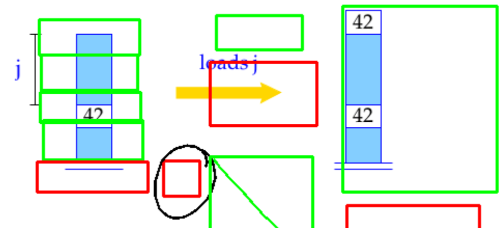
where $(V, b) = \rho_C(f)$

C = class of e_1

m = space for the actual parameters

if (next==0) return info;
else return next->last();

The instruction loads j loads relative to the stack pointer:



S[SP+1] = S[SP−j];
SP++;

43 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: **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 **new** $C\ (e_2, \ldots, e_n)\ \rho \quad = \quad$ loadc $|C|$
new
$\boxed{\text{initVirtual } C}$
$\text{code}_R\ e_n\ \rho$
...
$\text{code}_R\ e_2\ \rho$
loads m // loads relative to SP :-)
mark
loadc _C
call
pop m + 1

where m = space for the actual parameters.

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 :-)

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:

initVirtual $C \quad = \quad$ dup
loadc b_1 ; add
loadc a_1 ; store
pop
...
dup
loadc b_r ; add
loadc a_r ; store
pop

code_R **new** $C\ (e_2, \ldots, e_n)\ \rho \quad = \quad$ loadc $|C|$
new
initVirtual C
$\text{code}_R\ e_n\ \rho$
...
$\text{code}_R\ e_2\ \rho$
loads m // loads relative to SP :-)
mark
loadc _C
call
pop m $\boxed{+ 1}$

where m = space for the actual parameters.

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 :-)

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:

initVirtual $C \quad = \quad$ dup
loadc b_1 ; add
loadc a_1 ; store
pop
...
dup
loadc b_r ; add
loadc a_r ; store
pop

44 Defining Constructors

In general, a definition of a constructor for class C looks as follows:

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

Idea:

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

... in the Example:

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


44 Defining Constructors

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 :-)

390

40 Object Layout

info = x;

count ++ next = 0,

Idea:

- Only attributes and virtual member functions are stored inside the class !!
- The addresses of non-virtual or static member functions as well as of constructors can be resolved at compile-time :-)
- The fields of a sub-class are appended to the corresponding fields of the super-class ...

... in our Example:

info
next
last

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:

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

where m = space for the actual parameters.

The constructor is applied to the current object of the calling constructor!

45 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 intialized :-)

$$\text{code}_R\ C\,(e_2,\ldots,e_n)\ \rho \quad = \quad \begin{array}{l} \text{stalloc } |C| \\ \text{initVirtual } C \\ \text{code}_R\ e_n\ \rho \\ \ldots \\ \text{code}_R\ e_2\ \rho \\ \text{loads m} \\ \text{mark} \\ \text{loadc } \_C \\ \text{call} \\ \text{pop } m+2 \end{array}$$

where m = space for the actual parameters.

Note:

The instruction stalloc m is like malloc m but allocates on the stack :-)

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

SP = SP+m+1;

S[SP] = SP–m;

SP = SP+m+1;
S[SP] = SP−m;