Script **generated by TTT**

Title: Seidl: Virtual_Machines (14.06.2016)

Date: Tue Jun 14 10:23:22 CEST 2016

Duration: 90:52 min

Pages: 38

For every class C we assume that we are given an adress environment ρ_C .

ρ_C maps every identifier x visible inside C to its decorated relative address a . We distinguish:

global variable	(G, a)
local variable	(L, a)
attribute	(A, a)
virtual function	(V, b)
non-virtual function	(N, a)
static function	(S, a)

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.

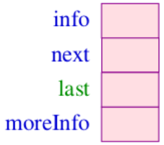
Idea (cont.)

- The fields of a sub-class are appended to the corresponding fields of the super-class.

Example

```
class mylist : list {
        int moreInfo;
    }
```

... results in:



For every class C we assume that we are given an adress environment ρ_C .

ρ_C maps every identifier x visible inside C to its decorated relative address a . We distinguish:

global variable	(G, a)
local variable	(L, a)
attribute	(A, a)
virtual function	(V, b)
non-virtual function	(N, a)
static function	(S, a)

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.

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

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

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

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

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

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

Accordingly, we introduce the abbreviated operations:

$$\text{loadm q} \ = \ \begin{array}{l} \text{loadr } -3 \\ \text{loadc q} \\ \text{add} \\ \text{load} \end{array}$$

$$\text{storem q} \ = \ \begin{array}{l} \text{loadr } -3 \\ \text{loadc q} \\ \text{add} \\ \text{store} \end{array}$$

Discussion

- Besides storing the current object pointer inside the stack frame, we could have additionally used a specific register *COP*.
- This register must 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.

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:

$$
\begin{array}{lll}
\text{code}_R\ e_1.f\,(e_2,\ldots,e_n)\ \rho & = & \text{code}_R\ e_n\ \rho \\
& & \ldots \\
& & \text{code}_R\ e_2\ \rho \\
& & \text{code}_L\ e_1\ \rho \\
& & \text{mark} \\
& & \text{loadc}\ \_f \\
& & \text{call} \\
& & \text{slide m}
\end{array}
$$

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

C = class of e_1

m = space for the actual parameters

Remark

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

A non-virtual function:

$$
\begin{array}{lll}
\text{code}_R\ e_1.f\,(e_2,\ldots,e_n)\ \rho & = & \text{code}_R\ e_n\ \rho \\
& & \ldots \\
& & \text{code}_R\ e_2\ \rho \\
& & \text{code}_L\ e_1\ \rho \\
& & \text{mark} \\
& & \text{loadc}\ \_f \\
& & \text{call} \\
& & \text{slide m}
\end{array}
$$

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

C = class of e_1

m = space for the actual parameters

Remark

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

A virtual function:

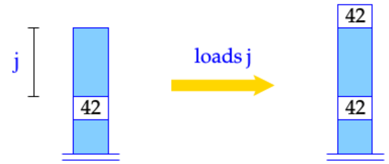
$$
\begin{array}{lll}
\text{code}_R\ e_1.f\,(e_2,\ldots,e_n)\ \rho & = & \text{code}_R\ e_n\ \rho \\
& & \ldots \\
& & \text{code}_R\ e_2\ \rho \\
& & \text{code}_L\ e_1\ \rho \\
& & \text{mark} \\
& & \text{loads 2} \\
& & \text{loadc}\ b \\
& & \text{add; load} \\
& & \text{call} \\
& & \text{slide m}
\end{array}
$$

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++;

A virtual function:

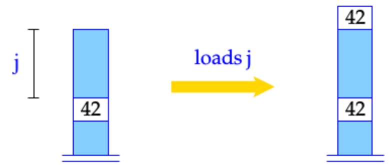
$$
\begin{aligned}
\text{code}_R\ e_1.f\ (e_2, \ldots, e_n)\ \rho \quad = \quad & \text{code}_R\ e_n\ \rho \\
& \ldots \\
& \text{code}_R\ e_2\ \rho \\
& \text{code}_L\ e_1\ \rho \\
& \text{mark} \\
& \text{loads 2} \\
& \text{loadc}\ b \\
& \text{add ; load} \\
& \text{call} \\
& \text{slide m}
\end{aligned}
$$

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

... 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 = $maxS + \text{m}$ where

$maxS$ = maximal depth of the local stack

m = space for the local variables

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
```

43 Calling Constructors

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

(1) directly: $C\ x\ (e_2, \ldots, e_n);$

(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** $C\ (e_2, \ldots, e_n)\ \rho$ =
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 + 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.

code_R **new** $C\ (e_2, \ldots, e_n)\ \rho$ =
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 + 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.

388

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 =
loadc a_1;
loads 1;
loadc b_1; add;
store; pop;
...
loadc a_r;
loads 1;
loadc b_r; add;
store; 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.

... 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	

Example

```
int count = 0;
class list {
        int info;
        class list * next;
        list (int x) {
                info = x;  count++;  next = null;
        }
        virtual int last () {
                if (next == null) return info;
                else return next → last ();
        }
}
```


... 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	

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:

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

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 constructor is called already at the declaration of x:

$$C\ x\ (e_2,\ldots,e_n);$$

Idea

- Push a reference to the memory block already allocated for x.
- Initialize that block.
- Pop the stack frame of the constructor together with the reference to x.

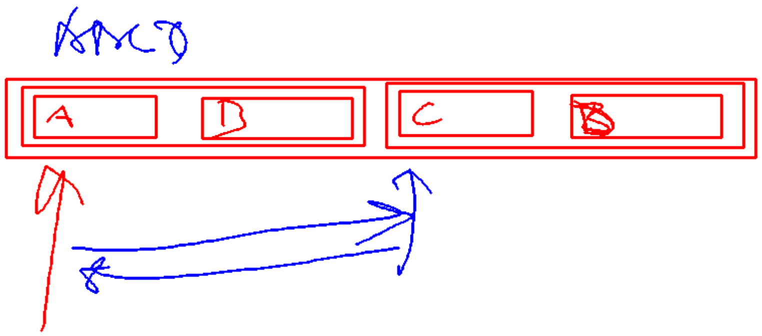
$$\begin{array}{lll} \text{code}_R\ C\ x\ (e_2,\ldots,e_n)\ \rho & = & \text{code}_L\ x\ \rho \\ & & \text{initVirtual } C \\ & & \text{code}_R\ e_n\ \rho \\ & & \ldots \\ & & \text{code}_R\ e_2\ \rho \\ & & \text{loads } m \\ & & \text{mark} \\ & & \text{loadc } \_C \\ & & \text{call} \\ & & \text{pop } m+2 \end{array}$$

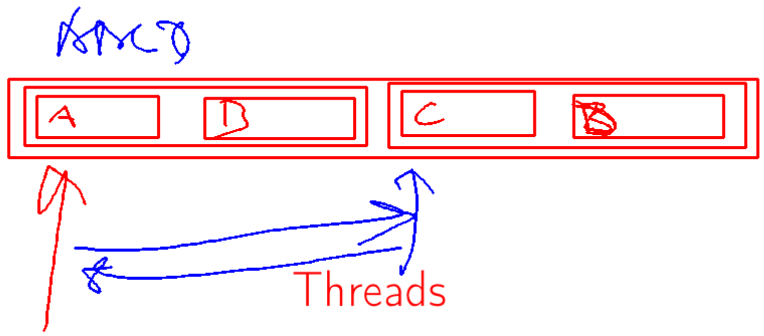
Pop

where m = space for the actual parameters.

$$\begin{array}{lll} \text{code}_R\ C\ x\ (e_2,\ldots,e_n)\ \rho & = & \text{code}_L\ x\ \rho \\ & & \text{initVirtual } C \\ & & \text{code}_R\ e_n\ \rho \\ & & \ldots \\ & & \text{code}_R\ e_2\ \rho \\ & & \text{loads } m \\ & & \text{mark} \\ & & \text{loadc } \_C \\ & & \text{call} \\ & & \text{pop } m+2 \end{array}$$

where m = space for the actual parameters.





46 The Language ThreadedC

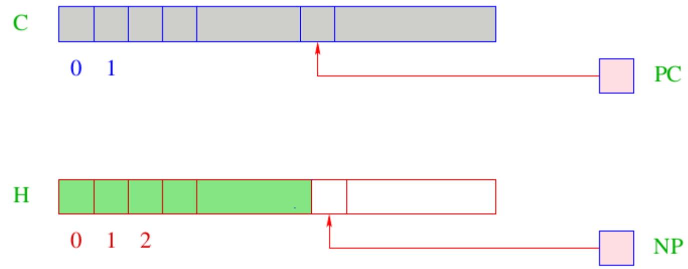
We extend C by a simple thread concept. In particular, we provide functions for:

- generating new threads: create();
- terminating a thread: exit();
- waiting for termination of a thread: join();
- mutual exclusion: lock(), unlock(); ...

In order to enable a parallel program execution, we extend the virtual machine (what else?)

47 Storage Organization

All threads share the same common code store and heap:

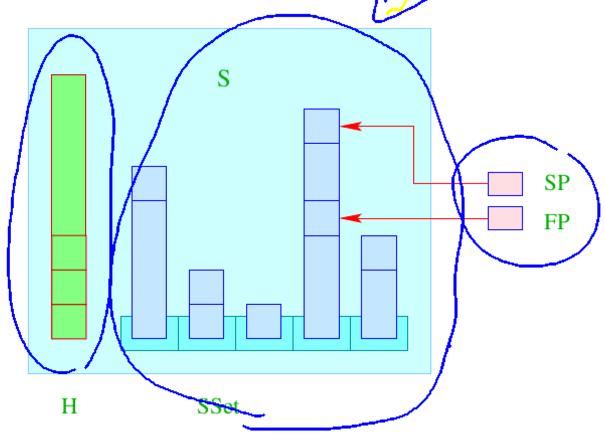


... similar to the CMa, we have:

C	=	Code Store – contains the CMa program; every cell contains one instruction;
PC	=	Program-Counter – points to the next executable instruction;
H	=	Heap – every cell may contain a base value or an address; the globals are stored at the bottom;
NP	=	New-Pointer – points to the first free cell.

For a simplification, we assume that the heap is stored in a separate segment. The function malloc() then fails whenever NP exceeds the topmost border.

Every thread on the other hand needs its own stack:



In constrast to the CMa, we have:

SSet	=	Set of Stacks – contains the stacks of the threads; every cell may contain a base value of an address;
S	=	common address space for heap and the stacks;
SP	=	Stack-Pointer – points to the current topmost ocupied stack cell;
FP	=	Frame-Pointer – points to the current stack frame.

Caveat

- If all references pointed into the heap, we could use separate address spaces for each stack.

  Besides SP and FP, we would have to record the number of the current stack.
- In the case of C, though, we must assume that all storage regions live within the same address space — only at different locations.

  SP Und FP then uniquely identify storage locations.
- For simplicity, we omit the extreme-pointer EP.