Script generated by TTT

Title: Seidl: Virtual_Machines (12.06.2013)

Date: Wed Jun 12 16:05:23 CEST 2013

Duration: 83:59 min

Pages: 42

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:

info	
next	
last	
moreInfo	

39 Object Layout

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	

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:

info	
next	
last	
moreInfo	

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 distingish:

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:

$$
\text{code}_{\text{L}}\; x\; \rho \;=\; \left\{ \begin{array}{ll} \text{loadr } {-3} & \text{if } \; x = \textbf{this} \\ \text{loadc } a & \text{if } \; \rho\, x = (G, a) \\ \text{loadr } a & \text{if } \; \rho\, x = (L, a) \\ \text{loadr } \cancel{1} \; -3 & \\ \text{loadc } a & \\ \text{add} & \text{if } \; \rho\, x = (A, a) \end{array} \right.
$$

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

Accordingly, we introduce the abbreviated operations:

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

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

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

$$
\text{code}_{\text{L}}\; x\; \rho \;=\; \left\{ \begin{array}{ll} \text{loadr } {-3} & \text{if } \; x = \textbf{this} \\ \text{loadc } a & \text{if } \; \rho\, x = (G, a) \\ \text{loadr } a & \text{if } \; \rho\, x = (L, a) \\ \text{loadr } 1 & \\ \text{loadc } a & \\ \text{add} & \text{if } \; \rho\, x = (A, a) \end{array} \right.
$$

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

Accordingly, we introduce the abbreviated operations:

```
loadm q   =  loadr −3
             loadc q
             add
             load

storem q  =  loadr −3
             loadc q
             add
             store
```


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 :-))

Accordingly, we introduce the abbreviated operations:

```
loadm q   =  loadr −3
             loadc q
             add
             load

storem q  =  loadr −3
             loadc q
             add
             store
```

363

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 :-))

364

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 $\mathbf{this}.f\ (e_2, \ldots, e_n)$:-)
- 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:

$$
\begin{aligned}
\mathrm{code_R}\ e_1.f\ (e_2, \ldots, e_n)\ \rho \quad = \quad & \mathrm{code_R}\ e_n\ \rho \\
& \ldots \\
& \mathrm{code_R}\ e_2\ \rho \\
& \mathrm{code_L}\ e_1\ \rho \\
& \mathrm{mark} \\
& \mathrm{loadc}\ \_f \\
& \mathrm{call} \\
& \mathrm{slide\ m}
\end{aligned}
$$

Pointer value

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:

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

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

C = class of e_1

m = space for the actual parameters

A non-virtual function:

$$
\begin{aligned}
\mathrm{code_R}\ e_1.f\ (e_2, \ldots, e_n)\ \rho \quad = \quad & \mathrm{code_R}\ e_n\ \rho \\
& \ldots \\
& \mathrm{code_R}\ e_2\ \rho \\
& \mathrm{code_L}\ e_1\ \rho \\
& \mathrm{mark} \\
& \boxed{\mathrm{loadc}\ \_f} \\
& \mathrm{call} \\
& \mathrm{slide\ m}
\end{aligned}
$$

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 :-)

366

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

A virtual function:

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

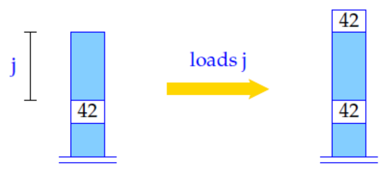
where $(V, b) = \rho_C(f)$

C = class of e_1

m = space for the actual parameters

367

The instruction loads j loads relative to the stack pointer:



S[SP+1] = S[SP–j];
SP++;

... in the Example:

this

The recursive call

next → last ()

(* 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
```


41 Defining Member Functions

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

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

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** ...

$$
\begin{array}{llll}
\text{code}_D\ d\ \rho = \_f: & \text{enter q} & // & \text{Setting the EP} \\
& \text{alloc m} & // & \text{Allocating the local variables} \\
& \text{code}\ ss\ \rho_1 & & \\
& \text{return} & // & \text{Leaving the function}
\end{array}
$$

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

$$
\begin{array}{llll}
\text{code}_D\ d\ \rho = \_f: & \text{enter q} & // & \text{Setting the EP} \\
& \text{alloc m} & // & \text{Allocating the local variables} \\
& \text{code}\ ss\ \rho_1 & & \\
& \text{return} & // & \text{Leaving the function}
\end{array}
$$

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

371

... 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: **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 ...

loadc |C|
new

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

```
malloc |C|
initVirtual C
codeR en ρ
...
codeR e2 ρ
loads m        // loads relative to SP  :-)
mark
loadc _C
call
pop m + 1
```

where m = space for the actual parameters.

Note:

Before calling the constructor, we initialize all fields of virtual functions.

The pointer to the object is copied into the frame by a new 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:

f_i

Then:

```
initVirtual C  =  dup
                  loadc b1 ;  add
                  loadc a1 ;  store
                  pop
                  ...
                  dup
                  loadc br ;  add
                  loadc ar ;  store
                  pop
```

43 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
```


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:

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

where m = space for the actual parameters.

Thus, 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 intialized :-)

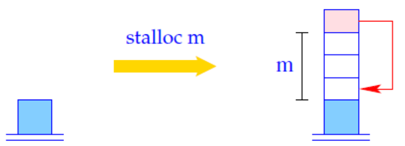
$$\begin{array}{lll}
\text{code}_R\ C\ (e_2, \ldots, e_n)\ \rho & = & \text{stalloc}\ |C| \\
& & \text{initVirtual}\ C \\
& & \text{code}_R\ e_n\ \rho \\
& & \ldots \\
& & \text{code}_R\ e_2\ \rho \\
& & \text{loads m} + 1 \\
& & \text{mark} \\
& & \text{loadc _C} \\
& & \text{call} \\
& & \text{pop m} + 1
\end{array}$$

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;

$$\begin{array}{lll}
\text{code}_R\ C\ (e_2, \ldots, e_n)\ \rho & = & \text{stalloc}\ |C| \\
& & \text{initVirtual}\ C \\
& & \text{code}_R\ e_n\ \rho \\
& & \ldots \\
& & \text{code}_R\ e_2\ \rho \\
& & \text{loads m} + 1 \\
& & \text{mark} \\
& & \text{loadc _C} \\
& & \text{call} \\
& & \text{pop m} + 2
\end{array}$$

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 :-)

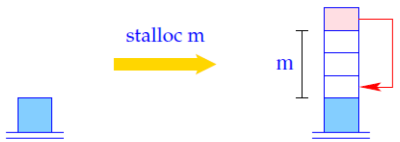
$$
\begin{array}{lcl}
\text{code}_R\ C\ (e_2, \ldots, e_n)\ \rho & = & \text{stalloc}\ |C| \\
 & & \text{initVirtual}\ C \\
 & & \text{code}_R\ e_n\ \rho \\
 & & \ldots \\
 & & \text{code}_R\ e_2\ \rho \\
 & & \text{loads}\ m+1 \\
 & & \text{mark} \\
 & & \text{loadc}\ \_C \\
 & & \text{call} \\
 & & \text{pop}\ m+1
\end{array}
$$

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;

Threads

① Interfaces

g_1 g_2

20

C : I_1, I_2

20

① Interfaces

g_1 g_2

20

$C: I_1, I_2$

20

① Interfaces

Threads

g_1 g_2

20

$C: I_1, I_2$

20