Script generated by TTT

Title: Seidl: Virtual_Machines (29.06.2015)

Date: Mon Jun 29 10:15:57 CEST 2015

Duration: 90:11 min

Pages: 40

Discussion:

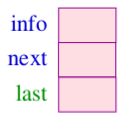
- We adopt the C++ perspective on classes and objects.
- We extend our implementation of C. In particular ...
- Classes are considered as extensions of structs. They may comprise:
  - ⇒ attributes, i.e., data fields;
  - ⇒ constructors;
  - ⇒ member functions which either are virtual, i.e., are called depending on the run-time type or non-virtual, i.e., called according to the static type of an object :-)
  - ⇒ static member functions which are like ordinary functions :-))
- We ignore visibility restrictions such as **public**, **protected** or **private** but simply assume general visibility.
- We ignore multiple inheritance :-)

40 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:



Example:

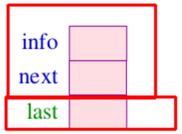
```
int count = 0;
class list {
        int info;
        class list * next;
        list (int x) {
                info = x; count++; next = null;
        }
        virtual int last () {
                if (next == null) return info;
                else return next → last ();
        }
}
```

40 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:



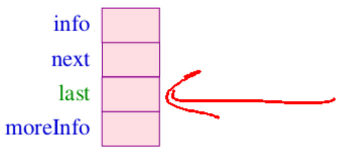
Idea (cont.):

- 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 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 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:

$e_0 . f(e_1, \ldots)$

$f(e_0, e_1, \ldots)$

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

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

Accordingly, we introduce the abbreviated operations:

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

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

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

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

Accordingly, we introduce the abbreviated operations:

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

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

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

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

Accordingly, we introduce the abbreviated operations:

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

377

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:

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

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

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:

$$\begin{array}{lcl} \text{code}_\text{R}\ e_1.f\ (e_2, \ldots, e_n)\ \rho & = & \text{code}_\text{R}\ e_n\ \rho \\ & & \ldots \\ & & \text{code}_\text{R}\ e_2\ \rho \\ & & \text{code}_\text{L}\ e_1\ \rho \\ & & \text{mark} \\ & & \text{loadc}\ \_f \\ & & \text{call} \\ & & \text{slide m} \end{array}$$

$$\begin{array}{ll} \text{where} & (F, \_f) = \rho_C(f) \\ & C = \text{class of } e_1 \\ & m = \text{space for the actual parameters} \end{array}$$

Note:

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

A non-virtual function:

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

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

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 non-virtual function:

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

N

where $(\cancel{F}, \_f) = \rho_C(f)$

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 :-)

380

A virtual function:

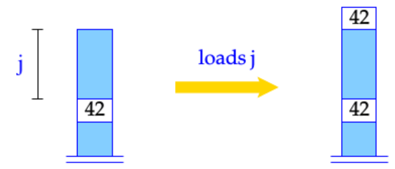
$$
\begin{array}{rcl}
\text{code}_R\ e_1.f\ (e_2, \ldots, e_n)\ \rho & = & \text{code}_R\ e_n\ \rho \\
& & \ldots \\
& & \text{code}_R\ e_2\ \rho \\
& & \text{code}_L\ e_1\ \rho \\
& & \text{mark} \\
& & \boxed{\begin{array}{l}\text{loads 2} \\ \text{loadc}\ b \\ \text{add ; load}\end{array}} \\
& &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

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++;

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** ...

... in the Example:

The recursive call

$$\text{next} \to \text{last}\;() \quad \equiv (*\boxed{\text{next}}).\text{last}()$$

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** ...

384

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

$$
\begin{array}{llll}
\text{where} & \text{q} & = & maxS + \text{m} \qquad \text{where} \\
 & maxS & = & \text{maximal depth of the local stack} \\
 & \text{m} & = & \text{space for the local variables} \\
 & \text{k} & = & \text{space for the formal parameters (including } \textbf{this}) \\
 & \rho_1 & = & \text{local address environment}
\end{array}
$$

... 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: $x = C\,(e_2, \ldots, e_n);$

(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** $C\,(e_2, \ldots, e_n)\ \rho$ = 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 :-)

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	

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 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 :-)

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

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;

Threads

Threads

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? :-)