Script generated by TTT

Title: Seidl: Virtual Machines (14.04.2014)

Date: Mon Apr 14 10:15:24 CEST 2014

Duration: 63:18 min

Pages: 43

Mo 12:00

We 10:00

Helmut Seidl

Virtual Machines

24.04 14:00

München

Summer 2014

22.04 ~~17:00~~

Mo 12:00

We 10:00

0 Introduction

Principle of Interpretation:

Program + Input ➡ **Interpreter** ➡ Output

24.04 14:00

Advantage: No precomputation on the program text ⟹ no/short startup-time

Disadvantages: Program parts are repeatedly analyzed during execution + less efficient access to program variables ⟹ slower execution speed.

22.04 ~~17:00~~

We define:

$$
\begin{array}{lcl}
\text{code}_R\ (e_1 + e_2)\ \rho & = & \text{code}_R\ e_1\ \rho \\
 & & \text{code}_R\ e_2\ \rho \\
 & & \text{add}
\end{array}
$$

... analogously for the other binary operators

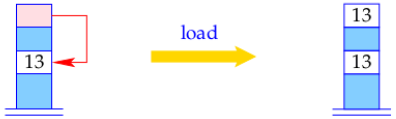
$$
\begin{array}{lcl}
\text{code}_R\ (-e)\ \rho & = & \text{code}_R\ e\ \rho \\
 & & \text{neg}
\end{array}
$$

... analogously for the other unary operators

$$
\begin{array}{lcl}
\text{code}_R\ q\ \rho & = & \text{loadc q} \\
\text{code}_L\ x\ \rho & = & \text{loadc}\ (\rho\, x) \\
 & \ldots &
\end{array}
$$

$$\text{code}_R\ x\ \rho \quad = \quad \text{code}_L\ x\ \rho \\ \text{load}$$

The instruction load loads the contents of the cell, whose address is on top of the stack.

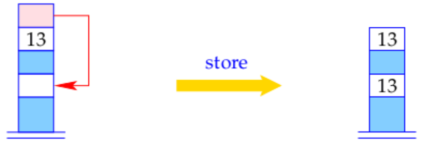


S[SP] = S[S[SP]];

$$\begin{aligned} \text{code}_R\ (x = e)\ \rho \quad = \quad & \text{code}_R\ e\ \rho \\ & \text{code}_L\ x\ \rho \\ & \text{store} \end{aligned}$$

store writes the contents of the second topmost stack cell into the cell, whose address in on top of the stack, and leaves the written value on top of the stack.

Note: this differs from the code generated by gcc ??



S[S[SP]] = S[SP-1];
SP--;

Simplification:

We only regard **switch**-statements of the following form:

$s \quad \equiv \quad$ **switch** (e) {
  case 0: ss_0 **break**;
  case 1: ss_1 **break**;
  $\vdots$
  case $k-1$: ss_{k-1} **break**;
  default: ss_k
}

s is then translated into the instruction sequence:

code s ρ =	$\text{code}_R\ e\ \rho$	C_0: code ss_0 ρ	B: jump C_0
	check 0 k B	jump D	...
		...	jump C_k
		C_k: code ss_k ρ	D: ...
		jump D	

- The Macro check 0 k B checks, whether the R-value of e is in the interval $[0, k]$, and executes an indexed jump into the table B
- The jump table contains direct jumps to the respective alternatives.
- At the end of each alternative is an unconditional jump out of the **switch**-statement.

```
check 0 k B  =  dup          dup          jumpi B
                loadc 0      loadc k   A: pop
                geq          le           loadc k
                jumpz A      jumpz A      jumpi B
```

- The R-value of e is still needed for indexing after the comparison. It is therefore copied before the comparison.
- This is done by the instruction dup.
- The R-value of e is replaced by k before the indexed jump is executed if it is less than 0 or greater than k.

Note:

- The jump table could be placed directly after the code for the Macro check. This would save a few unconditional jumps. However, it may require to search the **switch**-statement twice.
- If the table starts with u instead of 0, we have to decrease the R-value of e by u before using it as an index.
- If all potential values of e are definitely in the interval $[0, k]$, the macro check is not needed.

5 Storage Allocation for Variables

Goal:

Associate statically, i.e. at compile time, with each variable x a fixed (relative) address $\rho\, x$

Assumptions:

- variables of basic types, e.g. **int**, ... occupy one storage cell.
- variables are allocated in the store in the order, in which they are declared, starting at address 1.

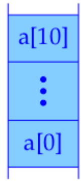
Consequently, we obtain for the declaration $d \equiv t_1\ x_1; \ldots t_k\ x_k;$ (t_i basic type) the address environment ρ such that

$$\rho\, x_i = i, \quad i = 1, \ldots, k$$

5.1 Arrays

Example: **int** [11] a;

The array a consists of 11 components and therefore needs 11 cells.
$\rho\, a$ is the address of the component $a[0]$.



We need a function sizeof (notation: $|\cdot|$), computing the space requirement of a type:

$$|t| = \begin{cases} 1 & \text{if } t \text{ basic} \\ k \cdot |t'| & \text{if } t \equiv t'[k] \end{cases}$$

Accordingly, we obtain for the declaration $d \equiv t_1\ x_1; \ldots t_k\ x_k;$

$$\begin{array}{lcll} \rho\, x_1 & = & 1 & \\ \rho\, x_i & = & \rho\, x_{i-1} + |t_{i-1}| & \text{for } i > 1 \end{array}$$

Since $|\cdot|$ can be computed at compile time, also ρ can be computed at compile time.

Task:

Extend $code_L$ and $code_R$ to expressions with accesses to array components.

Be $t[c]\ a;$ the declaration of an array a.

To determine the start address of a component $a[i]$, we compute $\rho\, a + |t| * (\textit{R-value of } i)$.

In consequence:

$$\begin{array}{lcl} code_L\ a[e]\ \rho & = & \text{loadc}\ (\rho\, a) \\ & & code_R\ e\ \rho \\ & & \text{loadc}\ |t| \\ & & \text{mul} \\ & & \text{add} \end{array}$$

... or more general:

$$\begin{array}{lcl} code_L\ e_1[e_2]\ \rho & = & code_R\ e_1\ \rho \\ & & code_R\ e_2\ \rho \\ & & \text{loadc}\ |t| \\ & & \text{mul} \\ & & \text{add} \end{array}$$

Remark:

- In C, an array is a pointer. A declared array a is a pointer-constant, whose R-value is the start address of the array.
- Formally, we define for an array e: $code_R\ e\ \rho = code_L\ e\ \rho$
- In C, the following are equivalent (as L-values):

$$2[a] \qquad a[2] \qquad a + 2$$

Normalization: Array names and expressions evaluating to arrays occur in front of index brackets, index expressions inside the index brackets.

5.2 Structures

In Modula and Pascal, structures are called Records.

Simplification:

Names of structure components are not used elsewhere. Alternatively, one could manage a separate environment ρ_{st} for each structure type st.

Be **struct** { **int** a; **int** b; } x; part of a declaration list.

- x has as relative address the address of the first cell allocated for the structure.
- The components have addresses relative to the start address of the structure. In the example, these are $a \mapsto 0, b \mapsto 1$.

5.2 Structures

In Modula and Pascal, structures are called Records.

Simplification:

Names of structure components are not used elsewhere.
Alternatively, one could manage a separate environment ρ_{st} for each structure type st.

Be **struct** { **int** a; **int** b; } x; part of a declaration list.

- x has as relative address the address of the first cell allocated for the structure.
- The components have addresses relative to the start address of the structure.
  In the example, these are $a \mapsto 0, b \mapsto 1$.

Let $t \equiv$ **struct** $\{t_1\ c_1; \ldots t_k\ c_k;\}$. We have

$$|t| = \sum_{i=1}^{k} |t_i|$$

$$\rho\, c_1 = 0 \quad \text{and}$$

$$\rho\, c_i = \rho\, c_{i-1} + |t_{i-1}| \quad \text{for } i > 1$$

We thus obtain:

$$\text{code}_L\ (e.c)\ \rho = \text{code}_L\ e\ \rho$$
$$\text{loadc}\ (\rho\, c)$$
$$\text{add}$$

Example:

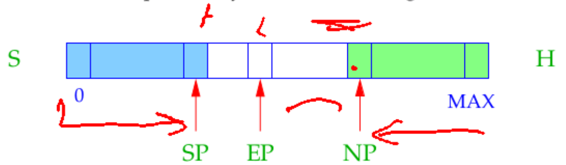
Be **struct** { **int** a; **int** b; } x; such that $\rho = \{x \mapsto 13, a \mapsto 0, b \mapsto 1\}$.
This yields:

$$\text{code}_L\ (x.b)\ \rho = \text{loadc } 13$$
$$\text{loadc } 1$$
$$\text{add}$$

6 Pointer and Dynamic Storage Management

Pointer allow the access to anonymous, dynamically generated objects, whose life time is not subject to the LIFO-principle.

$\Longrightarrow$ We need another potentially unbounded storage area H – the Heap.



NP $\widehat{=}$ New Pointer; points to the lowest occupied heap cell.

EP $\widehat{=}$ Extreme Pointer; points to the uppermost cell, to which SP can point (during execution of the actual function).

Idea:

- Stack and Heap grow toward each other in S, but must not collide. (Stack Overflow).
- A collision may be caused by an increment of SP or a decrement of NP.
- EP saves us the check for collision at the stack operations.
- The checks at heap allocations are still necessary.

What can we do with pointers (pointer values)?

- set a pointer to a storage cell,
- dereference a pointer, access the value in a storage cell pointed to by a pointer.

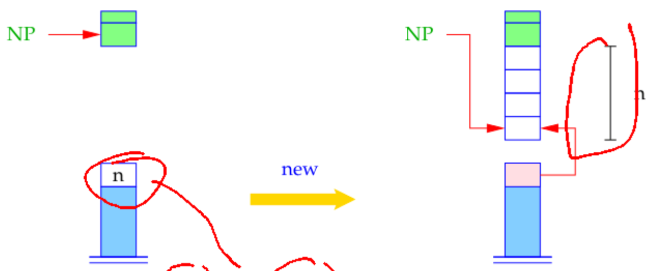
There a two ways to set a pointer:

(1) A call **malloc** (e) reserves a heap area of the size of the value of e and returns a pointer to this area:

$$\text{code}_R\ \textbf{malloc}\ (e)\ \rho \quad = \quad \text{code}_R\ e\ \rho$$
$$\qquad\qquad\qquad\qquad \text{new}$$

(2) The application of the address operator & to a variable returns a pointer to this variable, i.e. its address ($\widehat{=}$ L-value). Therefore:

$$\text{code}_R\ (\&e)\ \rho = \text{code}_L\ e\ \rho$$




```
if (NP - S[SP] ≤ EP)
    S[SP] = NULL;
else {
    NP = NP - S[SP];
    S[SP] = NP;
}
```

- NULL is a special pointer constant, identified with the integer constant 0.
- In the case of a collision of stack and heap the NULL-pointer is returned.

Dereferencing of Pointers:

The application of the operator $*$ to the expression e returns the contents of the storage cell, whose address is the R-value of e:

$$\text{code}_L\ (*e)\ \rho = \text{code}_R\ e\ \rho$$

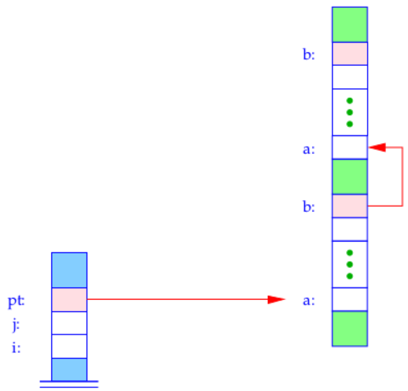
Example: Given the declarations

struct t { **int** $a[7]$; **struct** t $*b$; };
int i, j;
struct t $*pt$;

and the expression $((pt \to b) \to a)[i+1]$

Because of $e \to a \equiv (*e).a$ holds:

$$\text{code}_L\ (e \to a)\ \rho \quad = \quad \text{code}_R\ e\ \rho$$
$$\qquad\qquad\qquad\qquad\qquad \text{loadc}\ (\rho\ a)$$
$$\qquad\qquad\qquad\qquad\qquad \text{add}$$



Be $\rho = \{i \mapsto 1, j \mapsto 2, pt \mapsto 3, a \mapsto 0, b \mapsto 7\}$. Then:

$$
\begin{array}{lll}
\text{code}_L\ ((pt \to b) \to a)[i+1]\ \rho & & \\
= \text{code}_R\ ((pt \to b) \to a)\ \rho & = & \text{code}_R\ ((pt \to b) \to a)\ \rho \\
\quad \text{code}_R\ (i+1)\ \rho & & \text{loada } 1 \\
\quad \text{loadc } 1 & & \text{loadc } 1 \\
\quad \text{mul} & & \text{add} \\
\quad \text{add} & & \text{loadc } 1 \\
 & & \text{mul} \\
 & & \text{add}
\end{array}
$$

For arrays, their R-value equals their L-value. Therefore:

$$
\begin{array}{lllll}
\text{code}_R\ ((pt \to b) \to a)\ \rho & = & \text{code}_R\ (pt \to b)\ \rho & = & \text{loada } 3 \\
 & & \text{loadc } 0 & & \text{loadc } 7 \\
 & & \text{add} & & \text{add} \\
 & & & & \text{load} \\
 & & & & \text{loadc } 0 \\
 & & & & \text{add}
\end{array}
$$

In total, we obtain the instruction sequence:

loada 3	load	loada 1	loadc 1
loadc 7	loadc 0	loadc 1	mul
add	add	add	add

7 Conclusion

We tabulate the cases of the translation of expressions:

$$
\begin{array}{lll}
\text{code}_L\ (e_1[e_2])\ \rho & = & \text{code}_R\ e_1\ \rho \\
 & & \text{code}_R\ e_2\ \rho \\
 & & \text{loadc } |t| \\
 & & \text{mul} \\
 & & \text{add} \qquad\qquad \text{if } e_1 \text{ has type } t* \text{ or } t[\,]
\end{array}
$$

$$
\begin{array}{lll}
\text{code}_L\ (e.a)\ \rho & = & \text{code}_L\ e\ \rho \\
 & & \text{loadc } (\rho\, a) \\
 & & \text{add}
\end{array}
$$

7 Conclusion

We tabulate the cases of the translation of expressions:

$\text{code}_L\ (e_1[e_2])\ \rho$	=	$\text{code}_R\ e_1\ \rho$	
		$\text{code}_R\ e_2\ \rho$	
		loadc $\lvert t \rvert$	
		mul	
		add	if e_1 has type $t*$ or $t[]$
$\text{code}_L\ (e.a)\ \rho$	=	$\text{code}_L\ e\ \rho$	
		loadc $(\rho\, a)$	
		add	

$\text{code}_L\ (*e)\ \rho$	=	$\text{code}_R\ e\ \rho$	
$\text{code}_L\ x\ \rho$	=	loadc $(\rho\, x)$	
$\text{code}_R\ (\&e)\ \rho$	=	$\text{code}_L\ e\ \rho$	
$\text{code}_R\ e\ \rho$	=	$\text{code}_L\ e\ \rho$	if e is an array
$\text{code}_R\ (e_1 \Box e_2)\ \rho$	=	$\text{code}_R\ e_1\ \rho$	
		$\text{code}_R\ e_2\ \rho$	
		op	op instruction for operator '$\Box$'

$\text{code}_R\ q\ \rho$	=	loadc q	q constant
$\text{code}_R\ (e_1 = e_2)\ \rho$	=	$\text{code}_R\ e_2\ \rho$	
		$\text{code}_L\ e_1\ \rho$	
		store	
$\text{code}_R\ e\ \rho$	=	$\text{code}_L\ e\ \rho$	
		load	otherwise

Example: **int** $a[10], (*b)[10];$ with $\rho = \{a \mapsto 7, b \mapsto 17\}$.

For the statement: $*a = 5;$ we obtain:

$$\text{code}_L\ (*a)\ \rho = \text{code}_R\ a\ \rho = \text{code}_L\ a\ \rho = \text{loadc } 7$$

$\text{code}\ (*a = 5;)\ \rho$ = loadc 5
loadc 7
store
pop

As an exercise translate:

$$s_1 \equiv b = (\&a) + 2; \quad \text{and} \quad s_2 \equiv *(b + 3)[0] = 5;$$

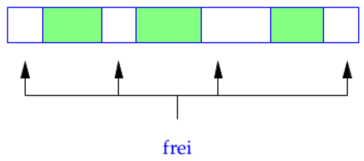
code $(s_1 s_2)\ \rho$ =

loadc 7		loadc 5	
loadc 2		loadc 17	
loadc 10	// *size of* **int**[10]	load	
mul	// *scaling*	loadc 3	
add		loadc 10	// *size of* **int**[10]
loadc 17		mul	// *scaling*
store		add	
pop	// *end of* s_1	store	
		pop	// *end of* s_2

8 Freeing Occupied Storage

Problems:

- The freed storage area is still referenced by other pointers (dangling references).
- After several deallocations, the storage could look like this (fragmentation):



Potential Solutions:

- Trust the programmer. Manage freed storage in a particular data structure (free list) ⟹ **malloc** or **free** my become expensive.
- Do nothing, i.e.:

  code **free** $(e);\ \rho$ = $\text{code}_R\ e\ \rho$
  pop

  ⟹ simple and (in general) efficient.
- Use an automatic, potentially "conservative" Garbage-Collection, which occasionally collects certainly inaccessible heap space.

9 Functions

The definition of a function consists of:

- a name by which it can be called;
- a specification of the formal parameters;
- a possible result type;
- a block of statements.

In C, we have:

$\text{code}_R\ f\ \rho$ = load c_f = start address of the code for f

⟹ Function names must be maintained within the address environment!

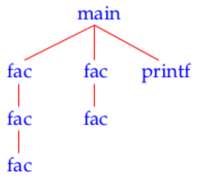
Example

```
int fac (int x) {
  if (x ≤ 0) return 1;
  else return x * fac(x − 1);
}
```

```
main () {
  int n;
  n = fac(2) + fac(1);
  printf ("%d", n);
}
```

At every point of execution, several instances (calls) of the same function may be active, i.e., have been started, but not yet completed.

The recursion tree of the example:



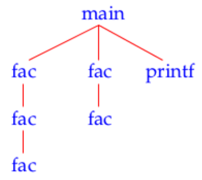
Example

```
int fac (int x) {
  if (x ≤ 0) return 1;
  else return x * fac(x − 1);
}
```

```
main () {
  int n;
  n = fac(2) + fac(1);
  printf ("%d", n);
}
```

At every point of execution, several instances (calls) of the same function may be active, i.e., have been started, but not yet completed.

The recursion tree of the example:



We conclude:

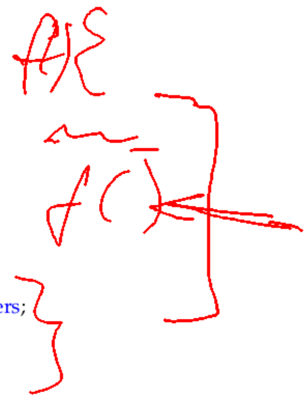
The formal parameters and local variables of the different calls of the same function (the instances) must be cept separate.

Idea

Allocate a dedicated memory block for each call of a function.

In sequential programming languages, these memory blocks may be maintained on a stack. Therefore, they are also called stack frames.

9 Functions



The definition of a function consists of:

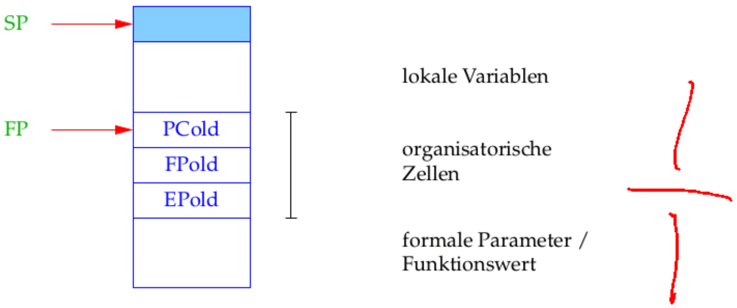
- a name by which it can be called;
- a specification of the formal parameters;
- a possible result type;
- a block of statements.

In C, we have:

$$\mathrm{code}_{\mathrm{R}}\ f\ \rho \quad = \quad \mathrm{load\ c\_}f \quad = \quad \text{start address of the code for } f$$

⟹ Function names must be maintained within the address environment!

9.1 Memory Organization for Functions



FP $\widehat{=}$ Frame Pointer; points to the last organizational cell and is used for addressing the formal parameters and local variables.

Caveat

- The local variables receive relative addresses $+1, +2, \ldots$.
- The formal parameters are placed below the organizational cells and therefore have negative addresses relative to FP :-)
- This organization is particularly well suited for function calls with variable number of arguments as, e.g., for `printf`.
- The memory block of parameters is recycled for storing the return value of the function :-))

Simplification The return value fits into a single memory cell.

9.2 Determining Address Environments

We distinguish two kinds of variables:

1. global/extern that are defined outside of functions;
2. local/intern/automatic (inkluding formal parameters) which are defined inside functions.

$\Longrightarrow$

The address environment ρ maps names onto pairs $(tag, a) \in \{G, L\} \times \mathbb{Z}$.

Caveat

- In general, there are further refined grades of visibility of variables.
- Different parts of a program may be translated relative to different address environments!

Example

```
0   int i;
    struct list {
      int info;
      struct list * next;
    } * l;

1   int ith (struct list * x, int i) {
      if (i ≤ 1) return x →info;
      else return ith (x →next, i − 1);
    }

2   main () {
      int k;
      scanf ("%d", &i);
      scanlist (&l);
      printf ("\n\t%d\n", ith (l,i));
    }
```