Script generated by TTT

Title: Seidl: Virtual_Machines (25.04.2016)

Date: Mon Apr 25 10:22:32 CEST 2016

Duration: 90:30 min

Pages: 40

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.

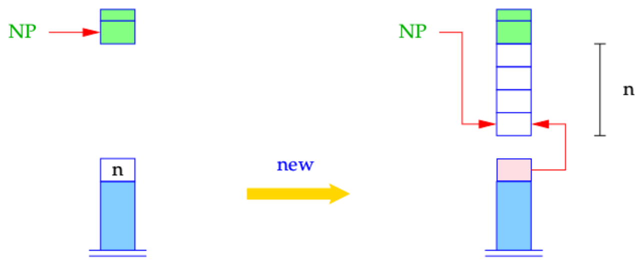
There a two ways to set a pointer:

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

$$\text{code}_R\ \textbf{malloc}\,(e)\ \rho \quad = \quad \begin{array}{l}\text{code}_R\ e\ \rho \\ \textbf{new}\end{array}$$

(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.

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:

$$\text{code}_R\ \textbf{malloc}\,(e)\ \rho \quad = \quad \begin{array}{l}\text{code}_R\ e\ \rho \\ \textbf{new}\end{array}$$

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

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

55

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:

$$\text{code}_\text{L}\ (*e)\ \rho = \text{code}_\text{R}\ e\ \rho$$

Example Given the declarations

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

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

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

$$\begin{aligned} \text{code}_\text{L}\ (e \to a)\ \rho \quad = \quad & \text{code}_\text{R}\ e\ \rho \\ & \text{loadc}\ (\rho\, a) \\ & \text{add} \end{aligned}$$

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:

$$\text{code}_\text{L}\ (*e)\ \rho = \text{code}_\text{R}\ e\ \rho$$

Example Given the declarations

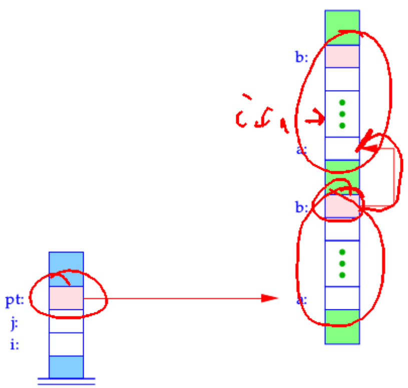
$\textbf{struct}\ t\ \{\ \textbf{int}\ a[7];\ \textbf{struct}\ t\ *b;\ \};$
$\textbf{int}\ i, j;$
$\textbf{struct}\ t\ *pt;$

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

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

$$\begin{aligned} \text{code}_\text{L}\ (e \to a)\ \rho \quad = \quad & \text{code}_\text{R}\ e\ \rho \\ & \text{loadc}\ (\rho\, a) \\ & \text{add} \end{aligned}$$

57



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

$$\begin{aligned} &\text{code}_\text{L}\ ((pt \to b) \to a)[i+1]\ \rho \\ = \quad & \text{code}_\text{R}\ ((pt \to b) \to a)\ \rho \quad = \quad && \text{code}_\text{R}\ ((pt \to b) \to a)\ \rho \\ & \text{code}_\text{R}\ (i+1)\ \rho && \text{loada } 1 \\ & \text{loadc } 1 && \text{loadc } 1 \\ & \text{mul} && \text{add} \\ & \text{add} && \text{loadc } 1 \\ & && \text{mul} \\ & && \text{add} \end{aligned}$$

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

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

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:

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

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

$$\begin{array}{lcll} \text{code}_R\ q\ \rho & = & \texttt{loadc q} & q \text{ constant} \\ \\ \text{code}_R\ (e_1 = e_2)\ \rho & = & \text{code}_R\ e_2\ \rho & \\ & & \text{code}_L\ e_1\ \rho & \\ & & \texttt{store} & \\ \\ \text{code}_R\ e\ \rho & = & \text{code}_L\ e\ \rho & \\ & & \texttt{load} & \text{otherwise} \end{array}$$

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

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

65

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

65

code $(s_1 s_2)\ \rho$ =

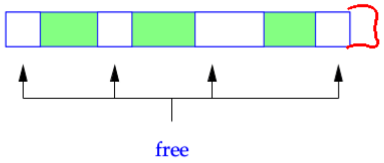
loadc 7		loadc 5	
loadc 2		loadc 17	
loadc 10	// *size of* **int**[10]	load	
mul	// *scaling*	loadc 3	
add		loadc 10	// *size of* **int**[10]
loadc 17		mul	// *scaling*
store		add	
pop	// *end of* s_1	store	
		pop	// *end of* s_2

65

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) $\Longrightarrow$ **malloc** or **free** my become expensive.
- Do nothing, i.e.:

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

$\Longrightarrow$ 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:

$$\text{code}_R\ f\ \rho \quad = \quad \text{loadc}\ \_f \quad = \quad \text{start address of the code for } f$$

$\Longrightarrow$ Function names must be maintained within the address environment!

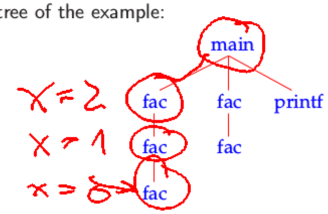
Example

```
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:



We conclude:

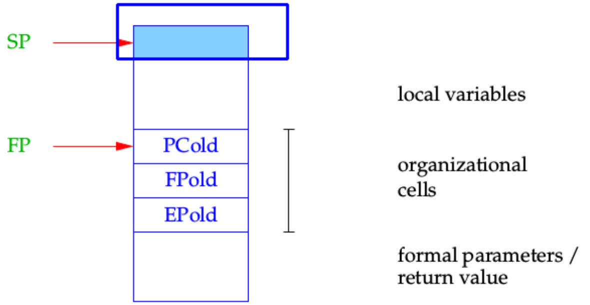
The formal parameters and local variables of the different calls of the same function (the instances) must be kept separate.

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.

9.1 Memory Organization for Functions



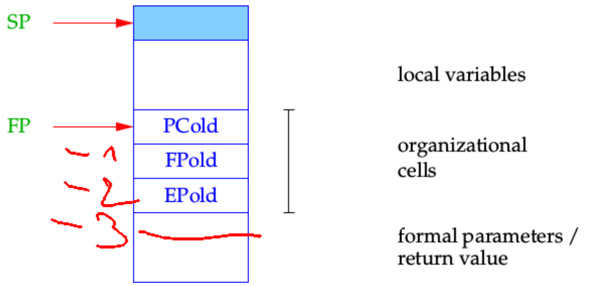
FP $\widehat{=}$ Frame Pointer; points to the last organizational cell and is used for addressing the formal parameters and local variables.

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.

9.1 Memory Organization for Functions



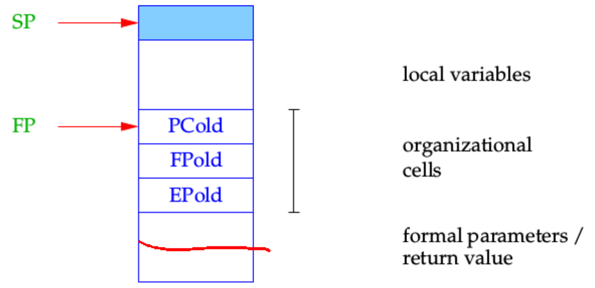
FP $\widehat{=}$ Frame Pointer; points to the last organizational cell and is used for addressing the formal parameters and local variables.

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.

9.1 Memory Organization for Functions



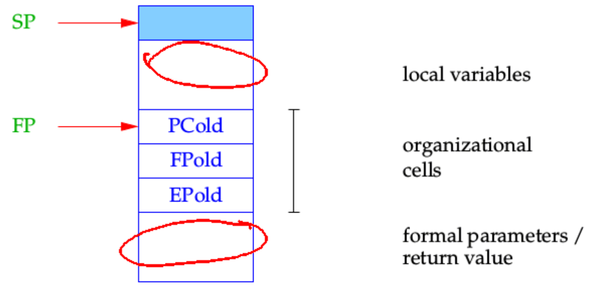
FP $\widehat{=}$ Frame Pointer; points to the last organizational cell and is used for addressing the formal parameters and local variables.

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.

9.1 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.

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.

Tasks of a Translator for Functions

- Generate code for the body of the function!
- Generate code for calls!

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));
    }
```

Address Environments Occurring in the Program

$\boxed{0}$ Before the Function Definitions:

$$\begin{array}{llll} \rho_0 : & i & \mapsto & (G,1) \\ & l & \mapsto & (G,2) \\ & & \dots & \end{array}$$

$\boxed{1}$ Inside of ith:

$$\begin{array}{llll} \rho_1 : & i & \mapsto & (L,-4) \\ & x & \mapsto & (L,-3) \\ & l & \mapsto & (G,2) \\ & \text{ith} & \mapsto & (G,\_\text{ith}) \\ & & \dots & \end{array}$$

Caveat

- The actual parameters are evaluated from right to left !!
- The first parameter resides directly below the organizational cells.
- For a prototype $\tau\ f(\tau_1\ x_1, \dots, \tau_k\ x_k)$ we define:

$$x_1 \mapsto (L, -2 - |\tau_1|) \qquad x_i \mapsto (L, -2 - |\tau_1| - \dots - |\tau_i|)$$

FP →

FP-3
FP-4

Caveat

- The actual parameters are evaluated from right to left !!
- The first parameter resides directly below the organizational cells.
- For a prototype $\tau\ f(\tau_1\ x_1, \dots, \tau_k\ x_k)$ we define:

$$x_1 \mapsto (L, -2 - |\tau_1|) \qquad x_i \mapsto (L, -2 - |\tau_1| - \dots - |\tau_i|)$$

77

Address Environments Occurring in the Program

$\boxed{0}$ Before the Function Definitions:

$$\begin{array}{llll} \rho_0 : & i & \mapsto & (G,1) \\ & l & \mapsto & (G,2) \\ & & \dots & \end{array}$$

$\boxed{1}$ Inside of ith:

$$\begin{array}{llll} \rho_1 : & i & \mapsto & (L,-4) \\ & x & \mapsto & (L,-3) \\ & l & \mapsto & (G,2) \\ & \text{ith} & \mapsto & (G,\_\text{ith}) \\ & & \dots & \end{array}$$

76

Example

$\boxed{0}$

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

$\boxed{1}$

```
int ith (struct list * x, int i) {
  if (i <= 1) return x ->info;
  else return ith (x ->next, i - 1);
}
```

$\boxed{2}$

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


Caveat

- The actual parameters are evaluated from right to left !!
- The first parameter resides directly below the organizational cells.
- For a prototype $\tau\ f(\tau_1\ x_1, \ldots, \tau_k\ x_k)$ we define:

$$x_1 \mapsto (L, -2 - |\tau_1|) \qquad x_i \mapsto (L, -2 - |\tau_1| - \ldots - |\tau_i|)$$

Caveat

- The actual parameters are evaluated from right to left !!
- The first parameter resides directly below the organizational cells.
- For a prototype $\tau\ f(\tau_1\ x_1, \ldots, \tau_k\ x_k)$ we define:

$$x_1 \mapsto (L, -2 - |\tau_1|) \qquad x_i \mapsto (L, -2 - |\tau_1| - \ldots - |\tau_i|)$$

$\boxed{2}$ Inside of main:

$$\begin{array}{lrcl} \rho_2 : & i & \mapsto & (G, 1) \\ & l & \mapsto & (G, 2) \\ & k & \mapsto & (L, 1) \\ & \text{ith} & \mapsto & (G, \_\text{ith}) \\ & \text{main} & \mapsto & (G, \_\text{main}) \\ & & \ldots & \end{array}$$

9.3 Calling/Entering and Exiting/Leaving Functions

Assume that f is the current function, i.e., the caller, and f calls the function g, i.e., the callee.

The code for the call must be distributed between the caller and the callee.

The distribution can only be such that the code depending on information of the caller must be generated for the caller and likewise for the callee.

Caveat

The space requirements of the actual paramters is only known to the caller ...

Actions when entering g:

1.	Evaluating the actual parameters		are part of f
2.	Saving of FP, EP	mark	
3.	Determining the start address of g		
4.	Setting of the new FP	call	
5.	Saving PC and Jump to the beginning of g		
6.	Setting of new EP	enter	are part of g
7.	Allocating of local variables	alloc	