Script generated by TTT

Title: Seidl: Virtual_Machines (27.04.2015)

Date: Mon Apr 27 10:20:16 CEST 2015

Duration: 85:09 min

Pages: 32

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:

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

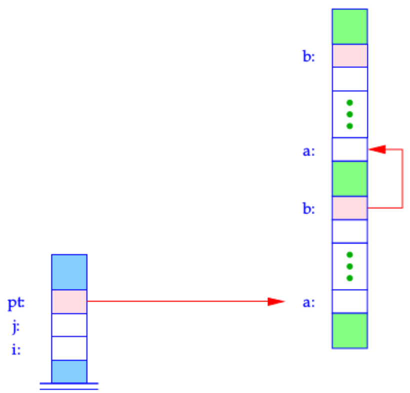
Example Given the declarations

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

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

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

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



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

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

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:

$$
\begin{array}{lcll}
\text{code}_L\,(e_1[e_2])\,\rho & = & \text{code}_R\,e_1\,\rho & \\
 & & \text{code}_R\,e_2\,\rho & \\
 & & \texttt{loadc}\,|t| & \\
 & & \texttt{mul} & \\
 & & \texttt{add} & \text{if } e_1 \text{ has type } t* \text{ or } t[\,] \\[1em]
\text{code}_L\,(e.a)\,\rho & = & \text{code}_L\,e\,\rho & \\
 & & \texttt{loadc}\,(\rho\,a) & \\
 & & \texttt{add} &
\end{array}
$$

$$
\begin{array}{lcll}
\text{code}_L\,(*e)\,\rho & = & \text{code}_R\,e\,\rho & \\[0.5em]
\text{code}_L\,x\,\rho & = & \texttt{loadc}\,(\rho\,x) & \\[0.5em]
\text{code}_R\,(\&e)\,\rho & = & \text{code}_L\,e\,\rho & \\[0.5em]
\text{code}_R\,e\,\rho & = & \text{code}_L\,e\,\rho & \text{if } e \text{ is an array} \\[0.5em]
\text{code}_R\,(e_1 \square e_2)\,\rho & = & \text{code}_R\,e_1\,\rho & \\
 & & \text{code}_R\,e_2\,\rho & \\
 & & \texttt{op} & \texttt{op} \text{ instruction for operator } `\square\text{'}
\end{array}
$$

$$
\begin{array}{lcll}
\text{code}_R\,q\,\rho & = & \texttt{loadc q} & q \text{ constant} \\[0.5em]
\text{code}_R\,(e_1 = e_2)\,\rho & = & \text{code}_R\,e_2\,\rho & \\
 & & \text{code}_L\,e_1\,\rho & \\
 & & \texttt{store} & \\[0.5em]
\text{code}_R\,e\,\rho & = & \text{code}_L\,e\,\rho & \\
 & & \texttt{load} & \text{otherwise}
\end{array}
$$

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:

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

$\text{code}\ (s_1 s_2)\ \rho$ =

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

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:

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

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

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

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

$$\begin{array}{lclclcl}
\text{code}_L\ (*a)\ \rho & = & \text{code}_R\ a\ \rho & = & \text{code}_L\ a\ \rho & = & \text{loadc } 7 \\
\text{code}\ (*a = 5;)\ \rho & = & \text{loadc } 5 & & & & \\
 & & \text{loadc } 7 & & & & \\
 & & \text{store} & & & & \\
 & & \text{pop} & & & &
\end{array}$$

As an exercise translate:

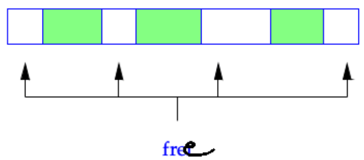
$$s_1 \equiv b = (\&a) + 2; \quad \text{and} \quad s_2 \equiv *(b+3)[0] = 5;$$

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

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

$\Longrightarrow$ simple and (in general) efficient.

- Use an automatic, potentially "conservative" Garbage-Collection, which occasionally collects certainly inaccessible heap space.

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.:

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

$\Longrightarrow$ 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!

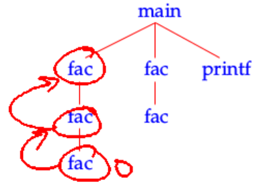
Example

```
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:

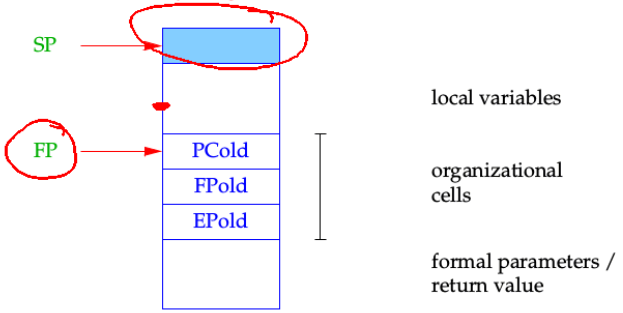
The formal parameters and local variables of the different calls of the same function (the instances) must be kept separate.

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



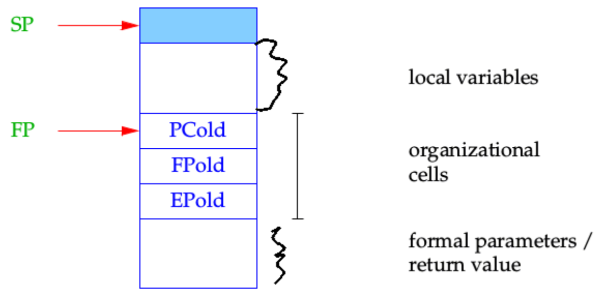
FP $\widehat{=}$ Frame Pointer; points to the last organizational cell and is used for addressing the formal parameters and local variables.

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



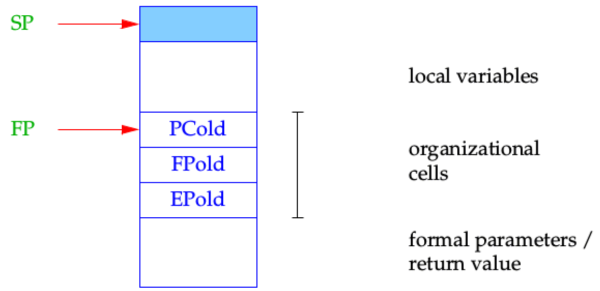
FP $\widehat{=}$ Frame Pointer; points to the last organizational cell and is used for addressing the formal parameters and local variables.

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



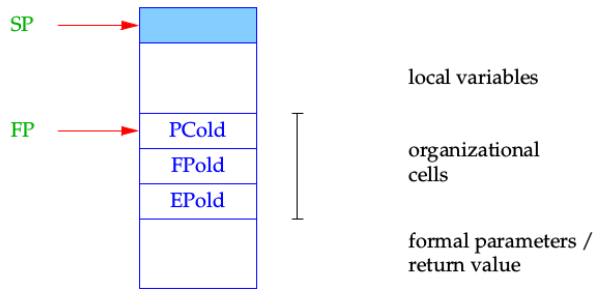
FP $\widehat{=}$ Frame Pointer; points to the last organizational cell and is used for addressing the formal parameters and local variables.

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 :-))

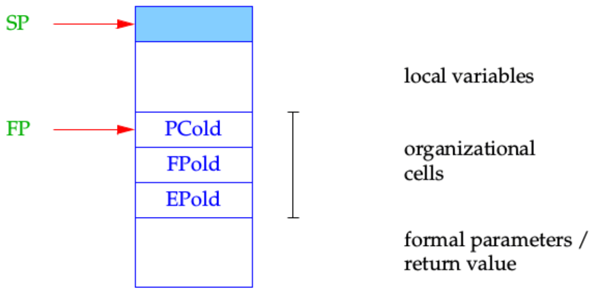
Simplification The return value fits into a single memory cell.

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.

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 :-))

  Simplificaiton: The return value fits into a single 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:**

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

$\boxed{1}$ **Inside of ith:**

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