Script generated by TTT

Title: Seidl: Virtual Machines (28.04.2014)

Date: Mon Apr 28 10:17:36 CEST 2014

Duration: 87:57 min

Pages: 48

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:

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

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

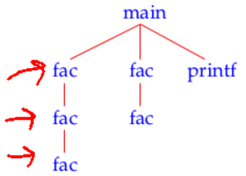
Example

```
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:

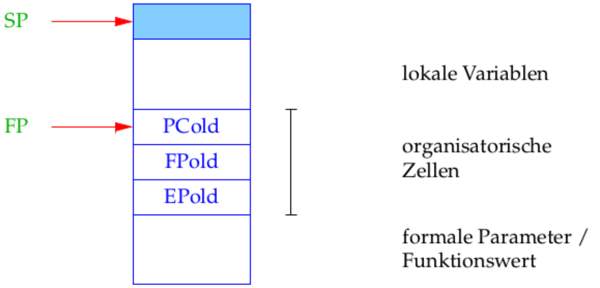
The formal parameters and local variables of the different calls of the same function (the instances) must be cept separate.

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.

9.1 Memory Organization for Functions



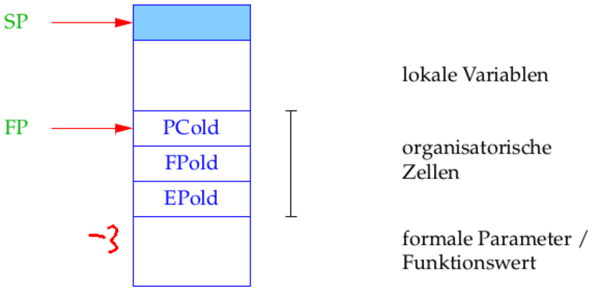
FP $\widehat{=}$ Frame Pointer; points to the last organizational cell and is used for addressing the formal parameters and local variables.

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.

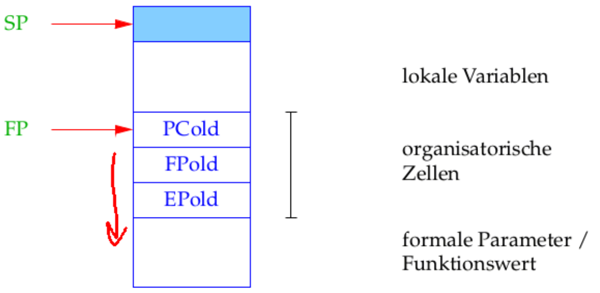
Caveat

- 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.

72

9.1 Memory Organization for Functions



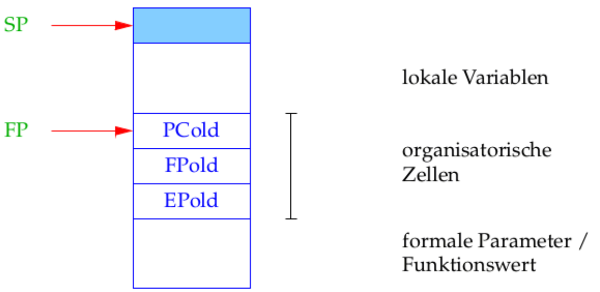
FP $\widehat{=}$ Frame Pointer; points to the last organizational cell and is used for addressing the formal parameters and local variables.

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.

71

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.

72

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:

$\boxed{0}$ Outside of the Function Definitions:

$$\begin{array}{rrcl} \rho_0 : & i & \mapsto & (G, 1) \\ & l & \mapsto & (G, 2) \\ & \text{ith} & \mapsto & (G, \_\text{ith}) \\ & \text{main} & \mapsto & (G, \_\text{main}) \\ & & \ldots & \end{array}$$

$\boxed{1}$ Inside of ith:

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

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:

$\boxed{0}$ Outside of the Function Definitions:

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

$\boxed{1}$ Inside of ith:

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

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:

$\boxed{0}$ Outside of the Function Definitions:

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

$\boxed{1}$ Inside of ith:

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

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:

[0] Outside of the Function Definitions:

ρ_0 :	i	$\mapsto$	$(G, 1)$
	l	$\mapsto$	$(G, 2)$
	ith	$\mapsto$	$(G, \_ith)$
	main	$\mapsto$	$(G, \_main)$
		...	

[1] Inside of ith:

ρ_1 :	i	$\mapsto$	$(L, -4)$
	x	$\mapsto$	$(L, -3)$
	l	$\mapsto$	$(G, 2)$
	ith	$\mapsto$	$(G, \_ith)$
	~~main~~	~~$\mapsto$~~	~~$(G, \_main)$~~
		...	

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:

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

Address Environments Occurring in the Program:

[0] Outside of the Function Definitions:

ρ_0 :	i	$\mapsto$	$(G, 1)$
	l	$\mapsto$	$(G, 2)$
	ith	$\mapsto$	$(G, \_ith)$
	main	$\mapsto$	$(G, \_main)$
		...	

[1] Inside of ith:

ρ_1 :	i	$\mapsto$	$(L, -4)$
	x	$\mapsto$	$(L, -3)$
	l	$\mapsto$	$(G, 2)$
	ith	$\mapsto$	$(G, \_ith)$
	main	$\mapsto$	$(G, \_main)$
		...	

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:

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

Address Environments Occurring in the Program:

$\boxed{0}$ Outside of the Function Definitions:

$$\begin{array}{rrcl} \rho_0 : & i & \mapsto & (G, 1) \\ & l & \mapsto & (G, 2) \\ & \text{ith} & \mapsto & (G, \_\text{ith}) \\ & \text{main} & \mapsto & (G, \_\text{main}) \\ & & \ldots & \end{array}$$

$\boxed{1}$ Inside of ith:

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

Caveat

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

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

$\boxed{2}$ Inside of main:

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

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 ...

f calls g

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	

Actions when terminating the call:

1. Storing of the return value	return
2. Restoring of the registers FP, EP, SP	
3. Jumping back into the code of f, i.e., Restauration of the PC	
4. Popping the stack	slide

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	

return

var. par.

Actions when terminating the call:

1. Storing of the return value
2. Restoring of the registers FP, EP ~~..~~
3. Jumping back into the code of f, i.e., Restauration of the PC
4. Popping the stack

$\left.\right\}$ return (items 1–3)

$\left.\right\}$ slide (item 4)

Accordingly, we obtain for a call to a function with at least one parameter and one return value:

$$
\begin{aligned}
\text{code}_R\ g(e_1, \ldots, e_n)\ \rho \quad = \quad & \text{code}_R\ e_n\ \rho \\
& \ldots \\
& \text{code}_R\ e_1\ \rho \\
& \text{mark} \\
& \text{code}_R\ g\ \rho \\
& \text{call} \\
& \boxed{\text{slide}\ (m-1)}
\end{aligned}
$$

where m is the size of the actual parameters.

Accordingly, we obtain for a call to a function with at least one parameter and one return value:

$$
\begin{aligned}
\text{code}_R\ g(e_1, \ldots, e_n)\ \rho \quad = \quad & \text{code}_R\ e_n\ \rho \\
& \ldots \\
& \text{code}_R\ e_1\ \rho \\
& \text{mark} \\
& \text{code}_R\ g\ \rho \\
& \text{call} \\
& \text{slide}\ (m-1)
\end{aligned}
$$

where m is the size of the actual parameters.

Remark

- Of every expression which is passed as a parameter, we determine the R-value $\Longrightarrow$ call-by-value passing of parameters.
- The function g may as well be denoted by an expression, dessen R-Wert die Anfangs-Adresse der aufzurufenden Funktion liefert ...

$*****g < g$

- Similar to declared arrays, function names are interpreted as constant pointes onto function code. Thus, the R-value of this pointer is the start address of the function.
- Caveat! For a variable **int** $(*)()\ g;$ the two calls

$$(*g)() \qquad \text{und} \qquad g()$$

  are equivalent! By means of normalization, the dereferencing of function pointers can be considered as redundant :-)
- During passing of parameters, these are copied.

Consequently,

$$\begin{array}{llll}
\text{code}_\text{R}\ f\ \rho & = & \text{loadc}\ (\rho\ f) & f \text{ name of a function} \\
\text{code}_\text{R}\ (*e)\ \rho & = & \text{code}_\text{R}\ e\ \rho & e \text{ function pointer} \\
\text{code}_\text{R}\ e\ \rho & = & \text{code}_\text{L}\ e\ \rho & \\
 & & \text{move k} & e \text{ a structure of size } k
\end{array}$$

where

Accordingly, we obtain for a call to a function with at least one parameter and one return value:

$$\begin{array}{lll}
\text{code}_\text{R}\ g(e_1, \ldots, e_n)\ \rho & = & \text{code}_\text{R}\ e_n\ \rho \\
 & & \ldots \\
 & & \text{code}_\text{R}\ e_1\ \rho \\
 & & \text{mark} \\
 & & \text{code}_\text{R}\ g\ \rho \\
 & & \text{call} \\
 & & \text{slide (m} - 1)
\end{array}$$

where m is the size of the actual parameters.

Actions when terminating the call:

1. Storing of the return value
2. Restoring of the registers FP, EP, SP
3. Jumping back into the code of f, i.e., Restauration of the PC

   } return

4. Popping the stack } slide

- Similar to declared arrays, function names are interpreted as constant pointes onto function code. Thus, the R-value of this pointer is the start address of the function.
- Caveat! For a variable **int** $(*)()\ g;$ the two calls

$$(*g)() \qquad \text{und} \qquad g()$$

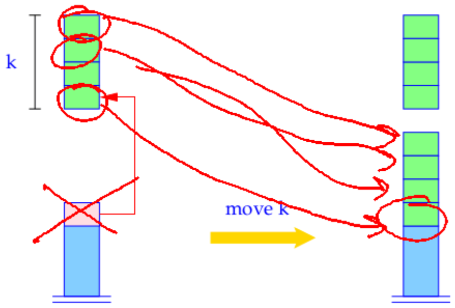
  are equivalent! By means of normalization, the dereferencing of function pointers can be considered as redundant :-)
- During passing of parameters, these are copied.

Consequently,

$$\begin{array}{llll}
\text{code}_\text{R}\ f\ \rho & = & \text{loadc}\ (\rho\ f) & f \text{ name of a function} \\
\text{code}_\text{R}\ (*e)\ \rho & = & \text{code}_\text{R}\ e\ \rho & e \text{ function pointer} \\
\text{code}_\text{R}\ e\ \rho & = & \text{code}_\text{L}\ e\ \rho & \\
 & & \text{move k} & e \text{ a structure of size } k
\end{array}$$

where

84

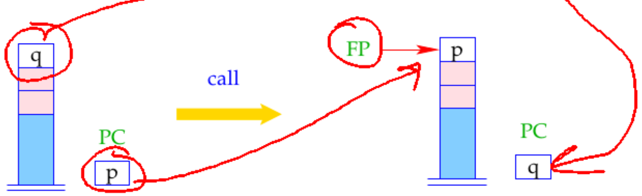



```
for (i = k-1; i≥0; i--)
    S[SP+i] = S[S[SP]+i];
SP = SP+k–1;
```


The instruction **mark** saves the registers FP and EP onto the stack.

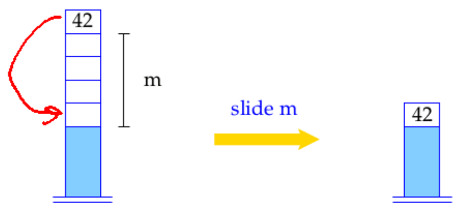

```
S[SP+1] = EP;
S[SP+2] = FP;
SP = SP + 2;
```


The instruction **call** saves the return address and sets FP and PC onto the new values.



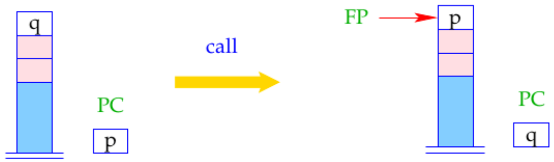

```
tmp = S[SP];
S[SP] = PC;
FP = SP;
PC = tmp;
```


The instruction **slide** copies the return values into the correct memory cell:




```
tmp = S[SP];
SP = SP-m;
S[SP] = tmp;
```

The instruction call saves the return address and sets FP and PC onto the new values.



tmp = S[SP];

S[SP] = PC;

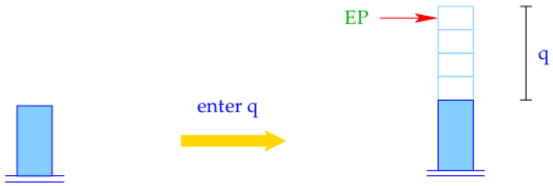
FP = SP;

PC = tmp;

Remark

- Of every expression which is passed as a parameter, we determine the R-value $\Longrightarrow$ call-by-value passing of parameters.
- The function g may as well be denoted by an expression, dessen R-Wert die Anfangs-Adresse der aufzurufenden Funktion liefert ...

The instruction enter q sets the EP to the new value. If not enough space is available, program execution terminates.



EP = SP + q;

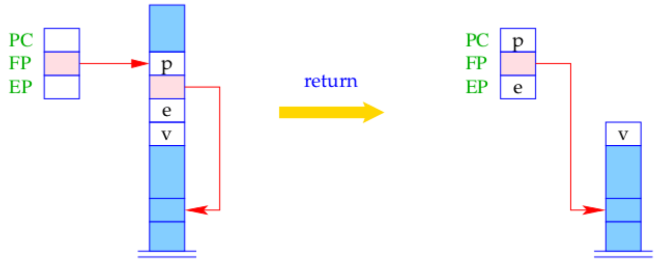
if (EP $\geq$ NP)

Error ("Stack Overflow");

The instruction alloc k allocates memory for locals on the stack.

SP = SP + k;

The instruction return pops the current stack frame. This means it restores the registers PC, EP and FP and returns the return value on top of the stack.



PC = S[FP]; EP = S[FP-2];

if (EP $\geq$ NP) Error ("Stack Overflow");

SP = FP-3; FP = S[SP+2];