Script generated by TTT

Title: Seidl: Virtual_Machines (24.04.2013)

Date: Wed Apr 24 16:00:15 CEST 2013

Duration: 90:36 min

Pages: 70

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:

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

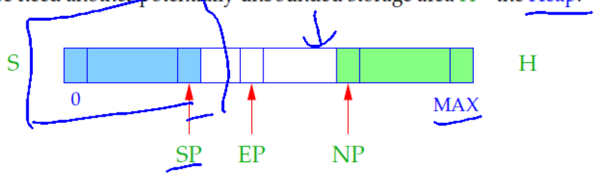
(2) The application of the address operator & to a variable returns a pointer to this variable, i.e. its address (≙ L-value). Therefore:

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

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.

⟹ We need another potentially unbounded storage area H – the Heap.



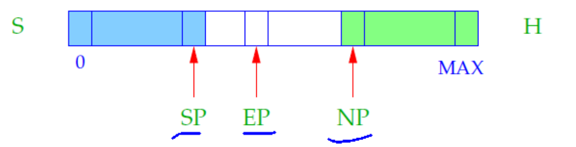
NP ≙ New Pointer; points to the lowest occupied heap cell.

EP ≙ Extreme Pointer; points to the uppermost cell, to which SP can point (during execution of the actual function).

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.

⟹ We need another potentially unbounded storage area H – the Heap.



NP ≙ New Pointer; points to the lowest occupied heap cell.

EP ≙ Extreme Pointer; points to the uppermost cell, to which SP can point (during execution of the actual function).

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:

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

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

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

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:

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

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:

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

(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:

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

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:

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

57

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:

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

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

$\text{code}_L\ ((pt \to b) \to a)[i+1]\ \rho$

$=$	$\text{code}_R\ ((pt \to b) \to a)\ \rho$	$=$	$\text{code}_R\ ((pt \to b) \to a)\ \rho$
	$\text{code}_R\ (i+1)\ \rho$		loada 1
	loadc 1		loadc 1
	mul		add
	add		loadc 1
			mul
			add

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:

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

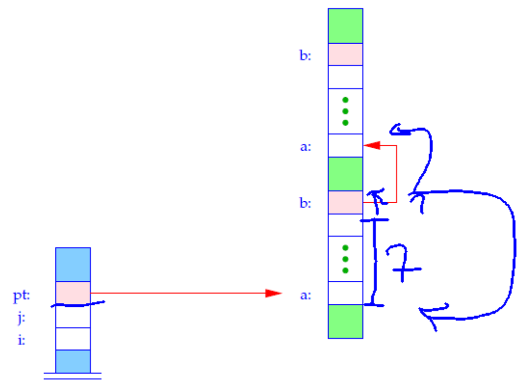
57

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

$\text{code}_L\ ((pt \to b) \to a)[i+1]\ \rho$

$=$	$\text{code}_R\ ((pt \to b) \to a)\ \rho$	$=$	$\text{code}_R\ ((pt \to b) \to a)\ \rho$
	$\text{code}_R\ (i+1)\ \rho$		loada 1
	loadc 1		loadc 1
	mul		add
	add		loadc 1
			mul
			add

59



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

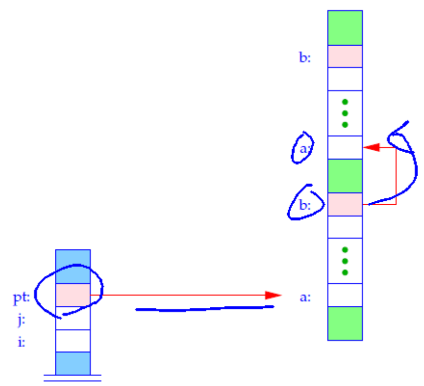
struct t { **int** $a[7]$; **struct** t $*b$; };
int i, j;
struct t $*pt$;

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

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

$$\text{code}_L\ (e \to a)\ \rho \;=\; \text{code}_R\ e\ \rho$$

loadc $(\rho\, a)$
add



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

$\text{code}_L\ ((pt \to b) \to a)[i+1]\ \rho$

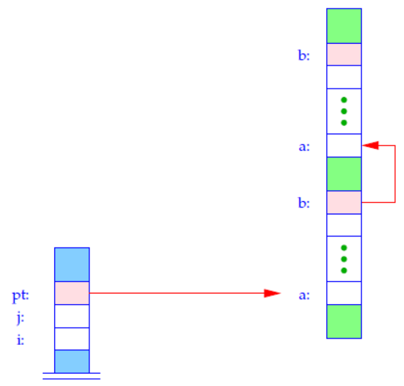
$=$ $\text{code}_R\ ((pt \to b) \to a)\ \rho$	$=$	$\text{code}_R\ ((pt \to b) \to a)\ \rho$
$\text{code}_R\ (i+1)\ \rho$		loada 1
loadc 1		loadc 1
mul		add
add		loadc 1
		mul
		add

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

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

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



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

$$\text{code}_L\ x\ \rho \quad = \quad \text{loadc}\ (\rho\ x)$$

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

$$\text{code}_R\ e\ \rho \quad = \quad \text{code}_L\ e\ \rho \qquad \text{if } e \text{ is an array}$$

$$\text{code}_R\ (e_1 \Box e_2)\ \rho \quad = \quad \begin{array}{l}\text{code}_R\ e_1\ \rho \\ \text{code}_R\ e_2\ \rho \\ \text{op}\end{array} \qquad \text{op instruction for operator '}\Box\text{'}$$

$$\text{code}_R\ q\ \rho \quad = \quad \text{loadc } q \qquad q \text{ constant}$$

$$\text{code}_R\ (e_1 = e_2)\ \rho \quad = \quad \begin{array}{l}\text{code}_R\ e_2\ \rho \\ \text{code}_L\ e_1\ \rho \\ \text{store}\end{array}$$

$$\text{code}_R\ e\ \rho \quad = \quad \begin{array}{l}\text{code}_L\ e\ \rho \\ \text{load}\end{array} \qquad \text{otherwise}$$

int [10] * b

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:

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

code $(s_1 s_2)\ \rho$ =

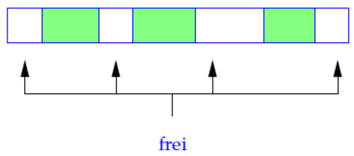
loadc 7		
loadc 2		
loadc 10	//	*size of* **int**[10]
mul	//	*scaling*
add		
loadc 17		
store		
pop	//	*end of* s_1
loadc 5		
loadc 17		
load		
loadc 3		
loadc 10	//	*size of* **int**[10]
mul	//	*scaling*
add		
store		
pop	//	*end of* s_2

code $(s_1 s_2)\ \rho$ =
loadc 7
loadc 2
loadc 10 // *size of* **int**[10]
mul // *scaling*
add
loadc 17
store
pop // *end of* s_1
loadc 5
loadc 17
load
loadc 3
loadc 10 // *size of* **int**[10]
mul // *scaling*
add
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:

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

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

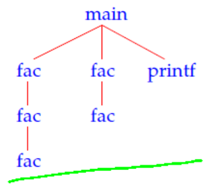
Example

```
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:

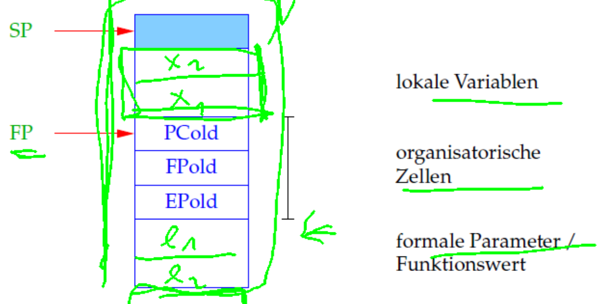
The formal parameters and local variables of the different calls of the same function (the instances) must be cept separate.

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



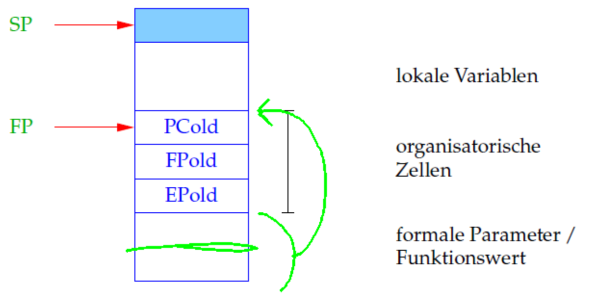
FP ≙ Frame Pointer; points to the last organizational cell and is used for addressing the formal parameters and local variables.

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



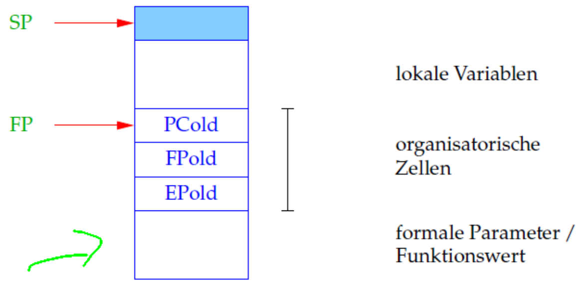
FP ≙ Frame Pointer; points to the last organizational cell and is used for addressing the formal parameters and local variables.

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



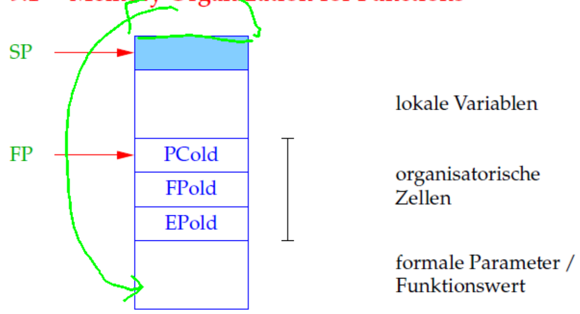
FP $\widehat{=}$ Frame Pointer; points to the last organizational cell and is used for addressing the formal parameters and local variables.

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 :-))

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:

0 Outside of the Function Definitions:

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

1 Inside of ith:

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

Address Environments Occurring in the Program:

0 Outside of the Function Definitions:

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

1 Inside of ith:

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

Address Environments Occurring in the Program:

0 Outside of the Function Definitions:

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

1 Inside of ith:

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

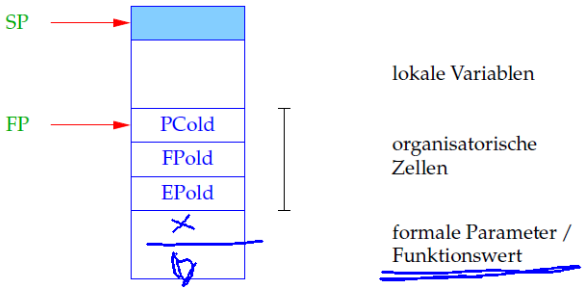
Example

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));
}
```

9.1 Memory Organization for Functions



FP ≙ Frame Pointer; points to the last organizational cell and is used for addressing the formal parameters and local variables.

Address Environments Occurring in the Program:

0 Outside of the Function Definitions:

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

1 Inside of ith:

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

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:

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

2 Inside of main:

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

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:

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

Actions when terminating the call:

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

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

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

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 ...

- 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

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

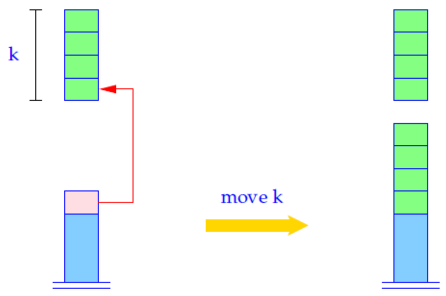
are equivalent! By means of normalization, the dereferencing of function pointers can be considered as redundant :-)

- During passing of parameters, these are copied.

Consequently,

$\text{code}_R\; f\; \rho$	=	loadc $(\rho\; f)$	f name of a function
$\text{code}_R\; (*e)\; \rho$	=	$\text{code}_R\; e\; \rho$	e function pointer
$\text{code}_R\; e\; \rho$	=	$\text{code}_L\; e\; \rho$	
		move k	e a structure of size k

where




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


- 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,

$\text{code}_R\; f\; \rho$	=	loadc $(\rho\; f)$	f name of a function
$\text{code}_R\; (*e)\; \rho$	=	$\text{code}_R\; e\; \rho$	e function pointer
$\text{code}_R\; e\; \rho$	=	$\text{code}_L\; e\; \rho$	
		move k	e a structure of size k

where

84

- 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,

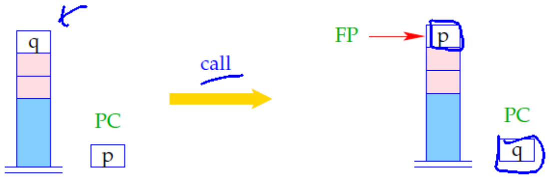
$\text{code}_R\ f\ \rho$	=	loadc $(\rho\ f)$	f name of a function
$\text{code}_R\ (*e)\ \rho$	=	$\text{code}_R\ e\ \rho$	e function pointer
$\text{code}_R\ e\ \rho$	=	$\text{code}_L\ e\ \rho$	
		move k	e a structure of size k

where

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

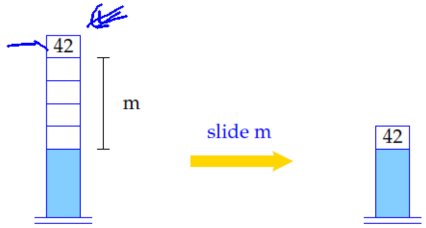

```
S[SP+1] = EP;
S[SP+2] = FP;
SP = SP + 2;
```


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;
```

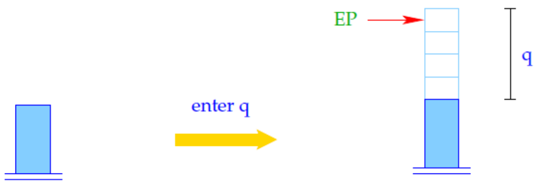
Accordingly, we translate a function definition:

code *t* f (*specs*){*V_defs* *ss*} ρ =

_f:	enter q	//	*initialize EP*
	alloc k	//	*allocate the local variables*
	code *ss* ρ_f		
	return	//	*return from call*

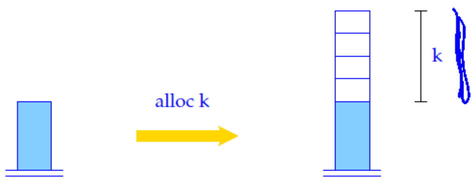
where	q	=	$max + k$	with
	max	=	maximal length of the local stack	
	k	=	size of the local variables	
	ρ_f	=	address environment for f	
			// *takes specs, V_defs* and ρ into account	

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



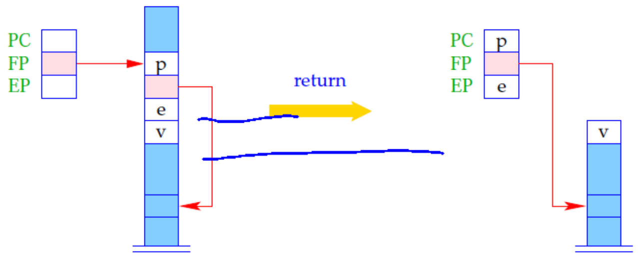

```
EP = SP + q;
if (EP ≥ NP)
    Error ("Stack Overflow");
```


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



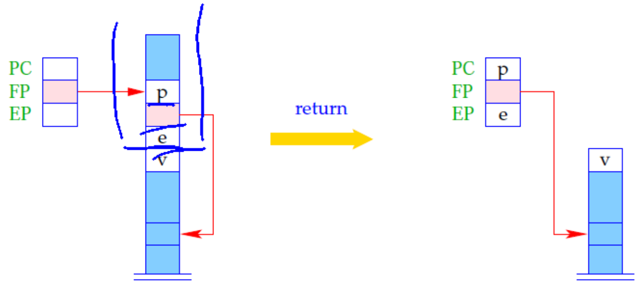
SP = SP + k;

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 ≥ NP) Error ("Stack Overflow");
SP = FP-3; FP = S[SP+2];
```

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];

9.4 Access to Variables, Formal Parameters and Returning of Values

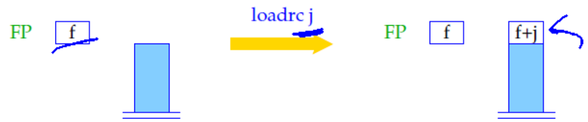
Accesses to local variables or formal parameters are relative to the current FP.

Accordingly, we modify code_L for names of variables.

For $\rho\, x = (tag, j)$ we define

$$\text{code}_L\; x\; \rho = \begin{cases} \text{loadc } j & tag = G \\ \text{loadrc } j & tag = L \end{cases}$$

The instruction loadrc j computes the sum of FP and j.



SP++;

S[SP] = FP+j;

As an optimization, we introduce analogously to loada j and storea j the new instructions loadr j and storer j :

loadr j = loadrc j
load

storer j = loadrc j;
store

The code for **return** e; corresponds to an assignment to a variable with relative address -3.

code **return** e; ρ = code$_R$ e ρ
storer -3
return

int f() {
return 7;
}

Example For function

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

we generate:

As an optimization, we introduce analogously to loada j and storea j the new instructions loadr j and storer j :

loadr j = loadrc j
load

storer j = loadrc j;
store

The code for **return** e; corresponds to an assignment to a variable with relative address -3.

code **return** e; ρ = code$_R$ e ρ
storer -3
return

Example For function

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

we generate: