Script **generated by TTT**

Title: Seidl: Virtual_machines (15.05.2012)

Date: Tue May 15 14:01:15 CEST 2012

Duration: 91:09 min

Pages: 45

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

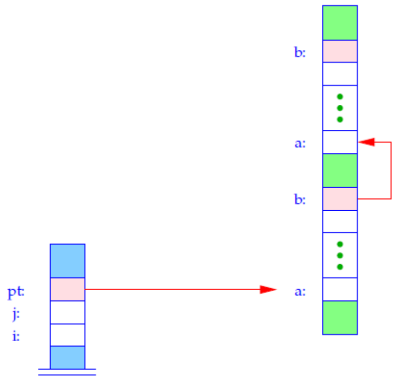
For the statement: $*a = 5;$ we obtain:

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

As an exercise translate:

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

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



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

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

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

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:

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

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

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

59

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

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

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

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

$$\begin{array}{llll}
\multicolumn{4}{l}{\text{code}_{\text{L}}\ ((pt \to b) \to a)[i+1]\ \rho} \\
= & \text{code}_{\text{R}}\ ((pt \to b) \to a)\ \rho & = & \text{code}_{\text{R}}\ ((pt \to b) \to a)\ \rho \\
& \text{code}_{\text{R}}\ (i+1)\ \rho & & \text{loada } 1 \\
& \text{loadc } 1 & & \text{loadc } 1 \\
& \text{mul} & & \text{add} \\
& \text{add} & & \text{loadc } 1 \\
& & & \text{mul} \\
& & & \text{add}
\end{array}$$

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

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

In total, we obtain the instruction sequence:

$$\begin{array}{llll}
\text{loada } 3 & \text{load} & \text{loada } 1 & \text{loadc } 1 \\
\text{loadc } 7 & \text{loadc } 0 & \text{loadc } 1 & \text{mul} \\
\text{add} & \text{add} & \text{add} & \text{add}
\end{array}$$

7 Conclusion

We tabulate the cases of the translation of expressions:

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

$$\begin{array}{llll}
\text{code}_{\text{L}}\ (*e)\ \rho & = & \text{code}_{\text{R}}\ e\ \rho & \\
\text{code}_{\text{L}}\ x\ \rho & = & \text{loadc } (\rho\, x) & \\
\text{code}_{\text{R}}\ (\&e)\ \rho & = & \text{code}_{\text{L}}\ e\ \rho & \\
\text{code}_{\text{R}}\ e\ \rho & = & \text{code}_{\text{L}}\ e\ \rho & \text{if } e \text{ is an array} \\
\text{code}_{\text{R}}\ (e_1 \Box e_2)\ \rho & = & \text{code}_{\text{R}}\ e_1\ \rho & \\
& & \text{code}_{\text{R}}\ e_2\ \rho & \\
& & \text{op} & \text{op instruction for operator `}\Box\text{'}
\end{array}$$

$\text{code}_L\ (*e)\ \rho$	$=$	$\text{code}_R\ e\ \rho$	
$\text{code}_L\ x\ \rho$	$=$	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
$\text{code}_R\ (e_1 \square e_2)\ \rho$	$=$	$\text{code}_R\ e_1\ \rho$	
		$\text{code}_R\ e_2\ \rho$	
		op	op instruction for operator '$\square$'

$\text{code}_R\ q\ \rho$	$=$	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** $a[10],\ *b;$ with $\rho = \{a \mapsto 7, b \mapsto 17\}$.

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

[0]

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

As an exercise translate:

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

$\text{code}\ (s_1 s_2)\ \rho$	$=$	loadc 7			loadc 5		
		loadc 2			loadc 17		
		loadc 10	//	*size of* **int**[10]	load		
		mul	//	*scaling*	loadc 3		
		add			loadc 1	//	*size of* **int**
		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 1	//	*size of* **int**
		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 1	//	*size of* **int**
		loadc 17			mul	//	*scaling*
		store			add		
		pop	//	*end of* s_1	store		
					pop	//	*end of* s_2

65

int (*b) [10];

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

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

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

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

As an exercise translate:

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

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**[illegible]
		loadc 17			mul	//	*scaling*
		store			add		
		pop	//	*end of* s_1	store		
					pop	//	*end of* s_2

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

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

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

As an exercise translate:

[0]

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

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

loadc 7		loadc 5	
loadc 2		loadc 17	
loadc 10	// *size of* **int**[10]	load	
mul	// *scaling*	loadc 3	
add		loadc 1	// *size of* **int**
loadc 17		mul	// *scaling*
store		add	
pop	// *end of* s_1	store	
		pop	// *end of* s_2

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

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

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

As an exercise translate:

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

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

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

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

As an exercise translate:

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

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

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

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

As an exercise translate:

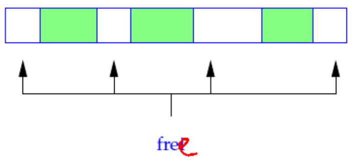
$$s_1 \equiv b = (\&a) + 2; \qquad \text{and} \qquad s_2 \equiv *(b+3) = 5;$$

$$\begin{array}{llllll}
\text{code}\ (s_1 s_2)\ \rho & = & \text{loadc } 7 & & \text{loadc } 5 & \\
 & & \text{loadc } 2 & & \text{loadc } 17 & \\
 & & \text{loadc } 10 & //\ \textit{size of } \textbf{int}[10] & \text{load} & \\
 & & \text{mul} & //\ \textit{scaling} & \text{loadc } 3 & \\
 & & \text{add} & & \text{loadc } 10 & //\ \textit{size of } \textbf{int}[\,] \\
 & & \text{loadc } 17 & & \text{mul} & //\ \textit{scaling} \\
 & & \text{store} & & \text{add} & \\
 & & \text{pop} & //\ \textit{end of } s_1 & \text{store} & \\
 & & & & \text{pop} & //\ \textit{end of } s_2
\end{array}$$

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.:

$$\begin{array}{lll}
\text{code}\ \textbf{free}\ (e);\ \rho & = & \text{code}_R\ e\ \rho \\
 & & \text{pop}
\end{array}$$

$\Longrightarrow$ 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.:

$$\begin{array}{lll} \text{code } \textbf{free}\,(e);\ \rho & = & \text{code}_R\ e\ \rho \\ & & \text{pop} \end{array}$$

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

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

g(f)

9 Functions

The definition of a function consists of

- a name, by which it can be called,
- a specification of the formal parameters;
- maybe a result type;
- a statement part, the body.

For C holds:

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

$\Longrightarrow$ Function names must also be managed in the address environment!

Example:

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

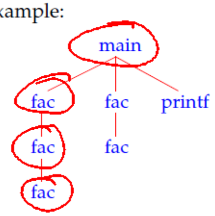
und

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

At any time during the execution, several instances of one function may exist, i.e., may have started, but not finished execution.

An instance is created by a call to the function.

The recursion tree in the example:



We conclude:

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

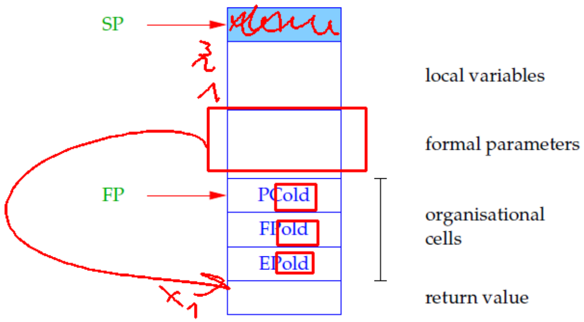
Idea:

Allocate a special storage area for each instance of a function.

In sequential programming languages these storage areas can be managed on a stack. They are therefore called Stack Frames.

$f(x_1, x_2, \ldots)$

9.1 Storage Organization for Functions



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

The caller must be able to continue execution in its frame after the return from a function. Therefore, at a function call the following values have to be saved into organizational cells:

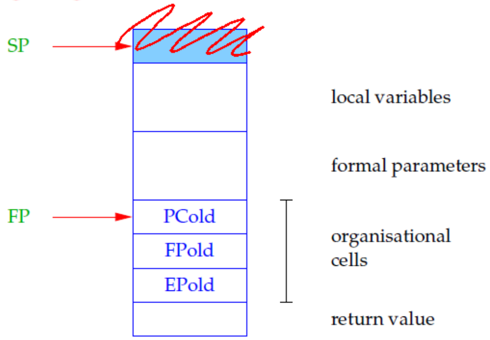
- the FP
- the continuation address after the call and
- the actual EP.

Simplification: The return value fits into one storage cell.

Translation tasks for functions:

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

9.1 Storage Organization for Functions



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

The caller must be able to continue execution in its frame after the return from a function. Therefore, at a function call the following values have to be saved into organizational cells:

- the FP
- the continuation address after the call and
- the actual EP.

Simplification: The return value fits into one storage cell.

Translation tasks for functions:

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

9.2 Computing the Address Environment

We have to distinguish two different kinds of variables:

1. globals, which are defined externally to the functions;
2. locals/automatic (including formal parameters), which are defined internally to the functions.

$$\Longrightarrow$$

The address environment ρ associates pairs $(tag, a) \in \{G, L\} \times \mathbb{N}_0$ with their names.

Note:

- There exist more refined notions of visibility of (the defining occurrences of) variables, namely nested blocks.
- The translation of different program parts in general uses different address environments!

9.2 Computing the Address Environment

We have to distinguish two different kinds of variables:

1. globals, which are defined externally to the functions;
2. locals/automatic (including formal parameters), which are defined internally to the functions.

$$\Longrightarrow$$

The address environment ρ associates pairs $(tag, a) \in \{G, L\} \times \mathbb{N}_0$ with their names.

Note:

- There exist more refined notions of visibility of (the defining occurrences of) variables, namely nested blocks.
- The translation of different program parts in general uses different address environments!

73

Example (1):

```
[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 environment at $\boxed{0}$

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

Example (2):

```
[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));
     }
```

$\boxed{1}$ inside of ith:

ρ_1 :	i	$\mapsto$	$(L, 2)$
	x	$\mapsto$	$(L, 1)$
	l	$\mapsto$	$(G, 2)$
	ith	$\mapsto$	$(G, \_ith)$
	~~main~~	~~$\mapsto$~~	~~$(G, \_main)$~~
		$\dots$	

Example (3):

$\boxed{0}$

```
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));
}
```

$\boxed{2}$ inside of main:

ρ_2 :	i	$\mapsto$	$(G, 1)$
	l	$\mapsto$	$(G, 2)$
	k	$\mapsto$	$(L, 1)$
	ith	$\mapsto$	$(G, \_\text{ith})$
	main	$\mapsto$	$(G, \_\text{main})$
		$\ldots$	

9.3 Calling/Entering and Leaving Functions

Be f the actual function, the Caller, and let f call the function g, the Callee.

The code for a function call has to be distributed among the Caller and the Callee:

The distribution depends on who has which information.

Actions upon calling/entering g:

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

(1.–5.: available in f; 6.–7.: available in g)

Actions upon leaving g:

1. Restoring the registers FP, EP, SP
2. Returning to the code of f, i.e. restoring the PC

} return

Altogether we generate for a call:

$$
\begin{aligned}
\text{code}_R\; g(e_1, \ldots, e_n)\; \rho \quad = \quad & \text{mark} \\
& \text{code}_R\; e_1\; \rho \\
& \ldots \\
& \text{code}_R\; e_m\; \rho \\
& \text{code}_R\; g\; \rho \\
& \text{call}\; \boxed{n}
\end{aligned}
$$

where n = space for the actual parameters

Note:

- Expressions occurring as actual parameters will be evaluated to their R-value $\Longrightarrow$ Call-by-Value-parameter passing.
- Function g can also be an expression, whose R-value is the start address of the function to be called ...

Actions upon calling/entering g:

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

(1.–5.: available in f; 6.–7.: available in g)

Actions upon leaving g:

1. Restoring the registers FP, EP, SP
2. Returning to the code of f, i.e. restoring the PC

} return

Altogether we generate for a call:

$$
\begin{array}{lll}
\mathrm{code}_R\ g(e_1, \ldots, e_n)\ \rho & = & \mathrm{mark} \\
 & & \mathrm{code}_R\ e_1\ \rho \\
 & & \ldots \\
 & & \mathrm{code}_R\ e_m\ \rho \\
 & & \mathrm{code}_R\ g\ \rho \\
 & & \mathrm{call\ n}
\end{array}
$$

where n = space for the actual parameters

Note:

- Expressions occurring as actual parameters will be evaluated to their R-value $\Longrightarrow$ Call-by-Value-parameter passing.
- Function g can also be an expression, whose R-value is the start address of the function to be called ...