Script generated by TTT

Title: Seidl: Virtual_Machines (28.04.2015)

Date: Tue Apr 28 10:16:38 CEST 2015

Duration: 88:21 min

Pages: 34

CHE survey

The Center for Higher Education Development CHE performs from 28.04.2015 another student survey on Master programs.

The results of the CHE survey flow directly into the CHE ranking of all German universities.

More information: www.in.tum.de/cheeng

CHE-Befragung

Ab dem 28.04.2015 führt das Zentrum für Hochschulentwicklung CHE wieder eine Befragung unter Studierenden von Master-Studiengängen durch.

Die Ergebnisse dieser Befragung fließen dann in das CHE-Ranking aller Universitäten in Deutschland ein.

Weitere Informationen unter www.in.tum.de/che

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:

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

$\boxed{1}$ Inside of ith:

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

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));
}
```


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));
}
```

75

Address Environments Occurring in the Program:

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

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

$\boxed{1}$ Inside of ith:

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

76

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));
      }
```


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:

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

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
2. Saving of FP, EP — mark
3. Determining the start address of g
4. Setting of the new FP
5. Saving PC and
   Jump to the beginning of g — call

(1.–5. are part of f)

6. Setting of new EP — enter
7. Allocating of local variables — alloc

(6.–7. are part of g)

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 — return
4. Popping the stack — slide

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

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

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, whose R-value provids the start address of the called function ...

- 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. g(e) ;

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

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.

- 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

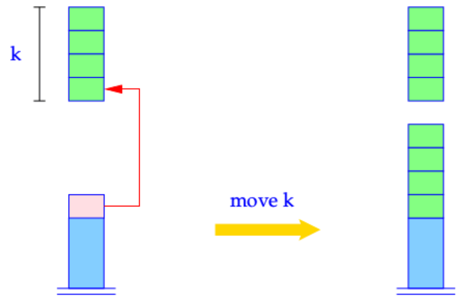
$$(*g)() \qquad \text{und} \qquad g()$$

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

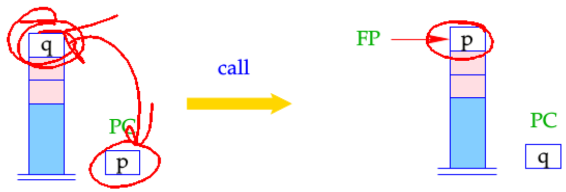



```
for (i = k-1; i≥0; i--)
    S[SP+i] = S[S[SP]+i];
SP = SP+k–1;
```

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

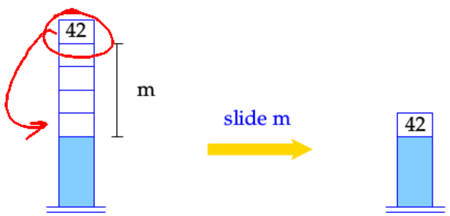
S[SP+1] = EP;
S[SP+2] = FP;
SP = SP + 2;

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 $(specs)\{V\_defs \quad ss\}$ 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

Accordingly, we translate a function definition:

code t f $(specs)\{V\_defs \quad ss\}$ ρ $=$

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

where q $=$ $max + \mathrm{k}$ with

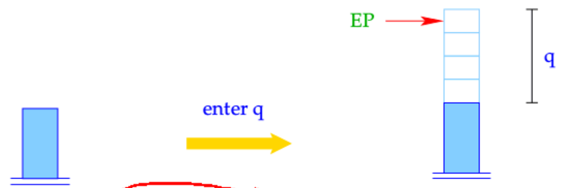
max $=$ maximal length of the local stack

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.

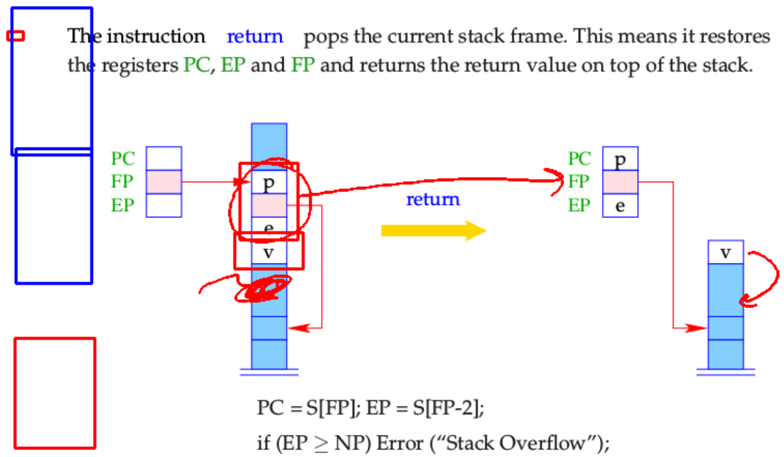


EP = SP + q;
if (EP $\geq$ NP)
Error (“Stack Overflow”);

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];

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 :

$\boxed{\text{loadr j}}$ = loadrc j
load

$\boxed{\text{storer j}}$ = loadrc j;
store

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

$$\text{code}\ \textbf{return}\ e;\ \rho \quad = \quad \begin{array}{l} \text{code}_R\ e\ \rho \\ \text{storer -3} \\ \text{return} \end{array}$$

Example For function

int fac (**int** x) {
  if ($x \leq 0$) **return** 1;
  else return $x *$ fac ($x - 1$);
}

we generate:

_fac:	enter q	loadc 1	A:	loadr -3		mul
	alloc 0	storer -3		loadr -3		storer -3
	loadr -3	return		loadc 1		return
	loadc 0	jump B		sub	B:	return
	leq			mark		
	jumpz A			loadc _fac		
				call		
				slide 0		

where $\rho_{\text{fac}} : x \mapsto (L, -3)$ and $\text{q} = 5$.

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

$$\text{code } \textbf{return}\ e;\ \rho \quad = \quad \text{code}_R\ e\ \rho$$

storer -3
return

Example For function

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

we generate:

_fac:	enter q	$\boxed{\text{loadc 1}}$	A:	loadr -3		mul
	alloc 0	$\boxed{\text{storer -3}}$		loadr -3		storer -3
	loadr -3	$\boxed{\text{return}}$		loadc 1		return
	loadc 0	~~jump B~~		sub	B:	~~return~~
	leq			mark		
	jumpz A			loadc _fac		
				call		
				slide 0		

where $\rho_{\text{fac}} : x \mapsto (L, -3)$ and $\text{q} = 5$.