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
Address Environments Occurring in the Program:

Before the Function Definitions:

\[ \rho_0 : \]
\[ i \rightarrow (G, 1) \]
\[ k \rightarrow (G, 2) \]

... 

Inside of \( \text{ith} \):

\[ \rho_1 : \]
\[ i \rightarrow (L, -4) \]
\[ x \rightarrow (L, -3) \]
\[ k \rightarrow (G, 2) \]
\[ \text{ith} \rightarrow (G, \_ith) \]

... 

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);
}
```

main () {
    int k;
    struct list * l;
    scanf ("%d", &i);
    scanf (&k);

    1 int ith (struct list * x, int i) {
        if (i <= 1) return x -> info;
        else return ith (x -> next, i - 1);
    }

    ... 

Address Environments Occurring in the Program:

Before the Function Definitions:

\[ \rho_0 : \]
\[ i \rightarrow (G, 1) \]
\[ k \rightarrow (G, 2) \]

... 

Inside of \( \text{ith} \):

\[ \rho_1 : \]
\[ i \rightarrow (L, -4) \]
\[ x \rightarrow (L, -3) \]
\[ k \rightarrow (G, 2) \]
\[ \text{ith} \rightarrow (G, \_ith) \]

... 

Example

```
0 int i;
struct list {
    int info;
    struct list * next;
} * l;

2 main () {
    int k;
    struct list * l;
    scanf ("%d", &i);
    scanf (&k);

    1 int ith (struct list * x, int i) {
        if (i <= 1) return x -> info;
        else return ith (x -> next, i - 1);
    }

    ... 
```
Example

```c
int i;
struct list {
  int info;
  struct list *next;
} x, y;

int ith (struct list *x, int i) {
  if (i <= 1) return x->info;
  else return ith (x->next, i - 1);
}
```

Caveat

- The actual parameters are evaluated from right to left (\textbullet{right to left} !!)
- The first parameter resides directly below the organizational cells (\textbullet{directly below})->
- For a prototype $\tau f(x_1, \ldots, x_n)$ we define:
  $x_1 \mapsto (L_n - 2 - |\tau_1|)$
  $x_i \mapsto (L_n - 2 - |\tau_1| - \ldots - |\tau_i|)$

Inside of `main`:

- $\rho_2$:
  - $i \mapsto (G_i, 1)$
  - $l \mapsto (G_i, 2)$
  - $k \mapsto (L_i, 1)$
  - $ith \mapsto (G_i, ith)$
  - $main \mapsto (G_i, main)$
  - ...

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 parameters is only known to the caller ...
**Actions when entering \( g \):**

1. **Evaluating the actual parameters**
2. **Saving of FP, EP**
3. Determining the start address of \( g \) are part of \( f \)
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
3. Jumping back into the code of \( f \), i.e., Restauration of the PC slide
4. Popping the stack

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

\[
\text{code}_{\text{e}}(g(e_1, \ldots, e_n), \rho) = \text{code}_{\text{e}}(e_n, \rho) \]

\[
\vdots
\]

\[
\text{code}_{\text{e}}(e_1, \rho) \]

\[
\text{mark}
\]

\[
\text{code}_{\text{e}}(g, \rho) \]

\[
\text{call}
\]

\[
\text{slide}(m - 1)
\]

where \( m \) is the size of the actual parameters.

**Remark**

- Of every expression which is passed as a parameter, we determine the R-value \( \implies \) call-by-value passing of parameters.
- The function \( g \) may as well be denoted by an expression, whose R-value provides the start address of the called function ...
• Similar to declared arrays, function names are interpreted as constant
  points onto function code. Thus, the R-value of this pointer is the start
  address of the function.
• Caveat! For a variable \( \text{int } (*)() \ g; \) the two calls
  \((*g)()\) and \(g()\)
are equivalent! By means of normalization, the dereferencing of function
pointers can be considered as redundant :)
• During passing of parameters, these are copied. Consequently,

\[
\text{code}_\text{R} f \, \rho = \text{loadc} (\rho f) \quad \text{f name of a function}
\]
\[
\text{code}_\text{R} (*e) \, \rho = \text{code}_\text{R} e \, \rho \quad \text{e function pointer}
\]
\[
\text{code}_\text{R} e \, \rho = \text{code}_\text{R} e \, \rho \quad \text{move } k \quad \text{e a structure of size } k
\]

where

\[\text{84}\]

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

\[
\text{code}_\text{R} g(e_1, \ldots, e_n) \, \rho = \text{code}_\text{R} e_n \, \rho
\]
\[
\ldots
\]
\[
\text{code}_\text{R} e_1 \, \rho
\]
\[
\text{mark}
\]
\[
\text{code}_\text{R} g \, \rho
\]
\[
\text{call}
\]
\[
\text{slide } (m - 1)
\]

where \( m \) is the size of the actual parameters.

\[\text{82}\]

• Similar to declared arrays, function names are interpreted as constant
  points onto function code. Thus, the R-value of this pointer is the start
  address of the function.
• Caveat! For a variable \( \text{int } (*)() \ g; \) the two calls
  \((*g)()\) and \(g()\)
are equivalent! By means of normalization, the dereferencing of function
pointers can be considered as redundant :)
• During passing of parameters, these are copied. Consequently,

\[
\text{code}_\text{R} f \, \rho = \text{loadc} (\rho f) \quad \text{f name of a function}
\]
\[
\text{code}_\text{R} (*e) \, \rho = \text{code}_\text{R} e \, \rho \quad \text{e function pointer}
\]
\[
\text{code}_\text{R} e \, \rho = \text{code}_\text{R} e \, \rho \quad \text{move } k \quad \text{e a structure of size } k
\]

where

\[\text{84}\]
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.

\[ \text{tmp} = S[SP]; \]
\[ S[SP] = PC; \]
\[ FP = SP; \]
\[ PC = \text{tmp}; \]

Accordingly, we translate a function definition:

```plaintext
code  t f (specs) \{ V_defs \ ss \} \rho^f =
    \_f:
        enter q \quad \text{// initialize EP}
        alloc k \quad \text{// allocate the local variables}
        code \rho \_f
        return \quad \text{// return from call}
```

where
\[ q = \max + k \text{ with} \]
\[ \max = \text{maximal length of the local stack} \]
\[ k = \text{size of the local variables} \]
\[ \rho_f = \text{address environment for } f \]
\[ \quad \text{// takes specs, V_defs and } \rho \text{ into account} \]
Accordingly, we translate a function definition:

```
code t f {specs} {V_defs ss} \[ \rho =
    f: enter q  // initialize EP
     alloc k   // allocate the local variables
      code ss \rho_f
    return  // return from call
```

where

- \( q = max + k \) with
- \( max \) = maximal length of the local stack
- \( k \) = size of the local variables
- \( \rho_f \) = address environment for \( f \)
  // takes specs, V_defs and \( \rho \) into account

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

The instruction \texttt{alloc k} allocates memory for locals on the stack.

\[ SP = SP + k; \]

The instruction \texttt{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.

\[ PC = S[FP]; \] \[ EP = S[FP-2]; \]
\[ if (EP \geq FP) \text{ Error ("Stack Overflow");} \]
\[ SP = FP-3; FP = S[SP+2]; \]
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\_i for names of variables.

For \[ x = (tag, j) \] we define

\[
\text{code}_i \ x \ \rho = \begin{cases} 
\text{loadc}_i & \text{tag} = G \\
\text{loadc}_j & \text{tag} = L 
\end{cases}
\]

As an optimization, we introduce analogously to \text{loada}_j and \text{storea}_j the new instructions \text{loadr}_j and \text{storer}_j:

\[
\text{loadr}_j = \text{loadc}_j \\
\text{storer}_j = \text{loadc}_j;
\]

The instruction \text{loadc}_j computes the sum of FP and \[ j \].

\[
\begin{array}{c}
\text{FP} \ f \\
\text{loadc}_j \\
\text{FP} \ f \ [j]
\end{array}
\]

\[ \text{SP}++; \]

\[ S[S] = \text{FP}+j; \]

The code for \text{return c}; \rho corresponds to an assignment to a variable with relative address \(-3\).

\[
\text{code return c}; \rho = \begin{cases} 
\text{codea}_\rho \\
\text{storer}_{-3} \text{return}
\end{cases}
\]

Example: For function

\[
\text{int fac (int x) |}
\]

\[
\text{if (x} \leq 0) \text{return 1;} \\
\text{else return x = fac (x - 1);} \\
\]

we generate:
The code for \texttt{return e;} corresponds to an assignment to a variable with relative address \(-3\).

\begin{align*}
\text{code return e; } & \rho = \\
& \text{code e } \rho \\
& \text{storer } -3 \\
& \text{return }
\end{align*}

Example For function
\begin{verbatim}
int fac (int x) |
  if (x \leq 0) return 1;
  else return x \times \text{ fac} (x - 1);
|
\end{verbatim}

we generate:

\begin{align*}
\text{\_fac: enter q &} \\
& \text{alloc 0} \\
& \text{loadc 1} \\
& \text{storer } -3 \\
& \text{return } \\
& \text{loadc } 0 \\
& \text{leq} \\
& \text{jumpz A}
\end{align*}

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