Script **generated by TTT**

Title: Seidl: Virtual_Machines (22.05.2012)

Date: Tue May 22 14:02:41 CEST 2012

Duration: 94:12 min

Pages: 24

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

Steps 1–5: } available in f

Steps 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

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.

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!

Altogether we generate for a call:

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

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 ...

- Function names are regarded as constant pointers to functions, similarly to declared arrays. The R-value of such a pointer is the start address of the function.
- For a variable **int** $(*)()\; g;$, the two calls

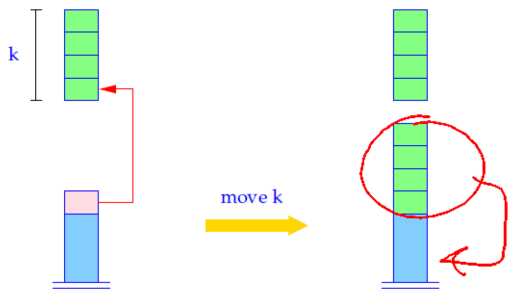
$$(*g)() \qquad \text{und} \qquad g()$$

are equivalent :-)
Normalization: Dereferencing of a function pointer is ignored.

- Structures are copied when they are passed as parameters.

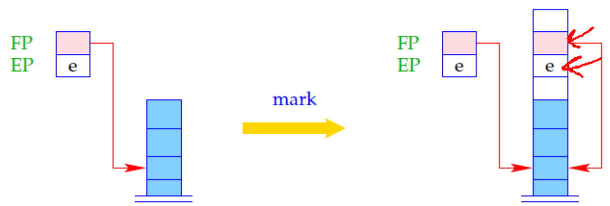
In consequence:

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



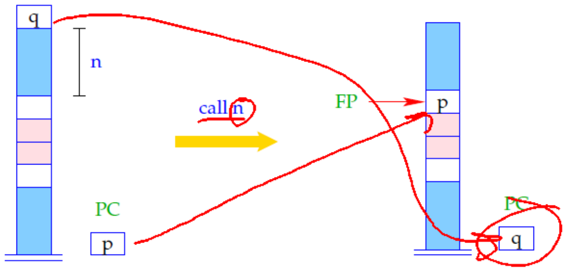
for (i = k-1; i≥0; i–)
S[SP+i] = S[S[SP]+i];
SP = SP+k–1;

The instruction mark allocates space for the return value and for the organizational cells and saves the FP and EP.



S[SP+2] = EP;
S[SP+3] = FP;
$\boxed{\text{SP = SP + 4;}}$

The instruction call n saves the continuation address and assigns FP, SP, and PC their new values.



FP = SP - n - 1;
S[FP] = PC;
PC = S[SP];
SP--;

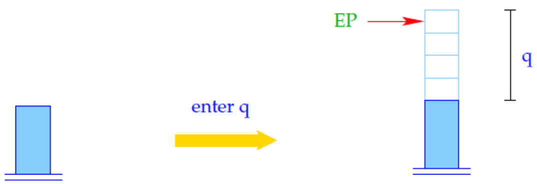
Correspondingly, we translate a function definition:

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

_f: enter q // *Setting the EP*
alloc k // *Allocating the local variables*
code ss ρ_f
return // *leaving the function*

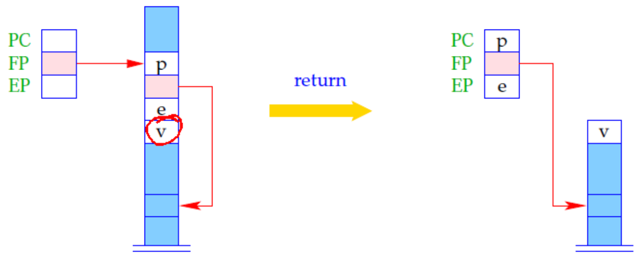
where t = return type of f with $|t| \leq 1$
q = $maxS$ + k where
$maxS$ = maximal depth of the local stack
k = space for the local variables
ρ_f = address environment for f
// *takes care of specs, V_defs* and ρ

The instruction enter q sets EP to its new value. Program execution is terminated if not enough space is available.



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

The instruction return pops the actual stack frame, i.e., it restores the registers PC, EP, SP, and FP and leaves 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];
FP - 1

9.4 Access to Variables and Formal Parameters, and Return of Values

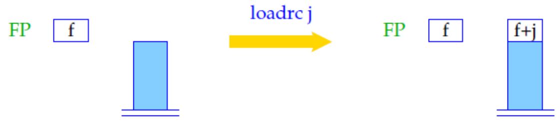
Local variables and formal parameters are addressed relative to the current FP.

We therefore modify code_L for the case of variable names.

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 one introduces the instructions loadr j and storer j .
This is analogous to loada j and storea 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 = $\text{code}_R\; e\; \rho$
storer -3
return

Example: For the function

maxh (e1 □ e2) =
max (e1,
maxh e1,
1 + max e2)

S ; SS

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

we generate:

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

where $\rho_{\text{fac}} : x \mapsto (L, 1)$ and $q = 1 + 4 + 2 = 7$.

As an optimization one introduces the instructions loadr j and storer j .
This is analogous to loada j and storea j.

$$\begin{array}{lll} \text{loadr j} & = & \text{loadrc j} \\ & & \text{load} \end{array}$$

$$\begin{array}{lll} \text{storer j} & = & \text{loadrc j} \\ & & \text{store} \end{array}$$

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

$$\begin{array}{lll} \text{code } \mathbf{return}\ e;\ \rho & = & \text{code}_{\mathrm{R}}\ e\ \rho \\ & & \text{storer -3} \\ & & \text{return} \end{array}$$

9.4 Access to Variables and Formal Parameters, and Return of Values

Local variables and formal parameters are addressed relative to the current FP.
We therefore modify code_{L} for the case of variable names.

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

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

10 Translation of Whole Programs

The state before program execution starts:

SP $= -1$ FP $=$ EP $= 0$ PC $= 0$ NP $=$ MAX $+1$

Be $p \equiv V\_defs\ F\_def_1 \ldots F\_def_n$, a program, where F_def_i defines a function f_i, of which one is named main.

The code for the program p consists of:

- Code for the function definitions F_def_i;
- Code for allocating the global variables;
- Code for the call of main();
- the instruction halt.

We thus define:

$$\begin{array}{rlll} \text{code } p\ \emptyset & = & & \text{enter } (k+6) \\ & & & \text{alloc } (k+1) \\ & & & \text{mark} \\ & & & \text{loadc \_main} \\ & & & \text{call } 0 \\ & & & \text{pop} \\ & & & \text{halt} \\ & & \_f_1: & \text{code } F\_def_1\ \rho \\ & & & \vdots \\ & & \_f_n: & \text{code } F\_def_n\ \rho \end{array}$$

$$\begin{array}{llll} \text{where} & \emptyset & \widehat{=} & \text{empty address environment;} \\ & \rho & \widehat{=} & \text{global address environment;} \\ & k & \widehat{=} & \text{space for global variables} \\ & \_main & \in & \{\_f_1, \ldots, \_f_n\} \end{array}$$

The Translation of Functional Programming Languages

11 The language PuF

We only regard a mini-language PuF ("Pure Functions").

We do not treat, as yet:

- Side effects;
- Data structures.

$(x\,y \Rightarrow x < y+6)$

A Program is an expression e of the form:

$$
\begin{array}{rcl}
e & ::= & b \mid x \mid (\Box_1\, e) \mid (e_1 \,\Box_2\, e_2) \\
 & \mid & (\textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2) \\
 & \mid & (e'\, e_0 \ldots e_{k-1}) \\
 & \mid & (\textbf{fn } x_0, \ldots, x_{k-1} \Rightarrow e) \\
 & \mid & (\textbf{let } x_1 = e_1; \ldots; x_n = e_n \textbf{ in } e_0) \\
 & \mid & (\textbf{letrec } x_1 = e_1; \ldots; x_n = e_n \textbf{ in } e_0)
\end{array}
$$

let fac = fn n => if ..

fac 6

An expression is therefore

- a basic value, a variable, the application of an operator, or
- a function-application, a function-abstraction, or
- a let-expression, i.e. an expression with locally defined variables, or
- a letrec-expression, i.e. an expression with simultaneously defined local variables.

let x = 6 in (let x = x + 1 in x) + x

let x = 7 in (x+1) + x

For simplicity, we only allow   as basic type.

if e' then e_1 else e_2 ≡ $(e\ e_1)\ e_2$

fn e e1 e2 = e e1 e2

Example:

true ≡ fn x y = x

false ≡ fn x y = y

The following well-known function computes the factorial of a natural number:

$$
\begin{array}{lcl}
\textbf{letrec}\ \text{fac} & = & \text{fn}\ x \Rightarrow \textbf{if}\ x \leq 1\ \textbf{then}\ 1 \\
 & & \textbf{else}\ x \cdot \text{fac}\ (x-1) \\
\textbf{in}\ \text{fac}\ 7 & &
\end{array}
$$

x and y ≡ fn x y t f =>

As usual, we only use the minimal amount of parentheses.

There are two Semantics:

CBV: Arguments are evaluated before they are passed to the function (as in SML);

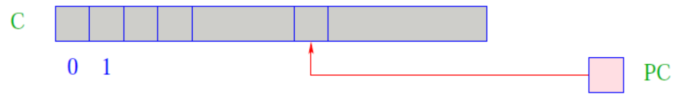
CBN: Arguments are passed unevaluated; they are only evaluated when their value is needed (as in Haskell).

let fst = fn x y = x

~~let dic = fn x => x/0~~

12 Architecture of the MaMa:

We know already the following components:



C = Code-store – contains the MaMa-program;
each cell contains one instruction;

PC = Program Counter – points to the instruction to be executed next;