Script generated by TTT

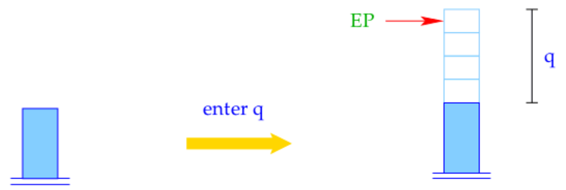
Title: Seidl: Virtual Machines (29.04.2014)

Date: Tue Apr 29 10:15:16 CEST 2014

Duration: 91:07 min

Pages: 26

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");

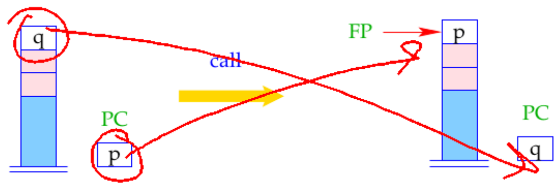
Accordingly, we translate a function definition:

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

The instruction call saves the return address and sets FP and PC onto the new values.

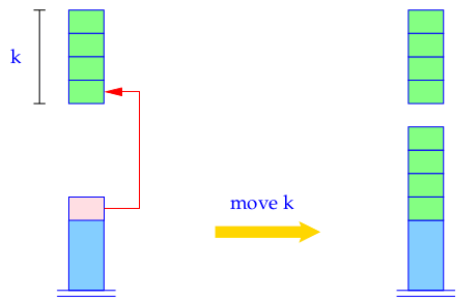


tmp = S[SP];
S[SP] = PC;
FP = SP;
PC = tmp;

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

$$\text{code}_R\ g(e_1, \ldots, e_n)\ \rho \quad = \quad \text{code}_R\ e_n\ \rho$$
$$\ldots$$
$$\text{code}_R\ e_1\ \rho$$
$$\text{mark}$$
$$\text{code}_R\ g\ \rho$$
$$\text{call}$$
$$\text{slide}\ (m-1)$$

where m is the size of the actual parameters.

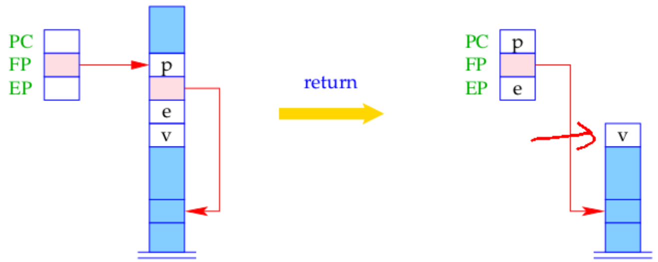


for (i = k-1; i≥0; i--)

S[SP+i] = S[S[SP]+i];

SP = SP+k–1;

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];

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

Example For function

```
int fac (int x) {
  if (x ≤ 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 q = 5.

0+1+1+3 = 5

10 Translation of Whole Programs

Before program execution, we have:

$$SP = -1 \qquad FP = EP = -1 \qquad PC = 0 \qquad NP = MAX$$

Let $p \equiv V\_defs\ \ F\_def_1 \dots F\_def_n$, denote a program where F_def_i is the definition of a function f_i of which one is called main .

The code for the program p consists of:

- code for the function definitions F_def_i;
- code for the allocation of global variables;
- code for the call of int main();
- the instruction halt which returns control to the operating system together with the value at address 0.

Then we define:

$$\begin{array}{rcl} \text{code}\ p\ \emptyset & = & \text{enter}\ (k+4) \\ & & \text{alloc}\ (k+1) \\ & & \text{mark} \\ & & \text{loadc \_main} \\ & & \text{call} \\ & & \text{slide}\ k \\ & & \text{halt} \\ \_f_1: & & \text{code}\ F\_def_1\ \rho \\ & & \vdots \\ \_f_n: & & \text{code}\ F\_def_n\ \rho \end{array}$$

where $\emptyset \ \widehat{=}\ $ empty address environment;
$\rho \ \widehat{=}\ $ global address environment;
$k \ \widehat{=}\ $ size of the global variables

10 Translation of Whole Programs

Before program execution, we have:

$$SP = -1 \qquad FP = EP = -1 \qquad PC = 0 \qquad NP = MAX$$

Let $p \equiv V\_defs\ \ F\_def_1 \dots F\_def_n$, denote a program where F_def_i is the definition of a function f_i of which one is called main .

The code for the program p consists of:

- code for the function definitions F_def_i;
- code for the allocation of global variables;
- code for the call of int main();
- the instruction halt which returns control to the operating system together with the value at address 0.

Then we define:

$$\begin{array}{rcl} \text{code}\ p\ \emptyset & = & \text{enter}\ (k+4) \\ & & \text{alloc}\ (k+1) \\ & & \text{mark} \\ & & \text{loadc \_main} \\ & & \text{call} \\ & & \text{slide}\ k \\ & & \text{halt} \\ \_f_1: & & \text{code}\ F\_def_1\ \rho \\ & & \vdots \\ \_f_n: & & \text{code}\ F\_def_n\ \rho \end{array}$$

where $\emptyset \ \widehat{=}\ $ empty address environment;
$\rho \ \widehat{=}\ $ global address environment;
$k \ \widehat{=}\ $ size of the global variables

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.

11 The language PuF

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

We do not treat, as yet:

- Side effects;
- Data structures.

let rec fac x = ☐ in fac 5

A program is an expression e of the form:

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

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 **let-rec**-expression, i.e. an expression with simultaneously defined local variables.

For simplicity, we only allow `int` as basic type.

Example:

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

$$\begin{aligned} \textbf{let rec } \mathrm{fac} \quad &= \quad \textbf{fun } x \to \textbf{if } x \leq 1 \textbf{ then } 1 \\ & \qquad \textbf{else } x \cdot \mathrm{fac}\ (x - 1) \\ \textbf{in } \mathrm{fac}\ 7 & \end{aligned}$$

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).

A program is an expression e of the form:

$$\begin{aligned} e \quad ::= \quad & b \mid x \mid (\Box_1\ e) \mid (e_1\ \Box_2\ e_2) \\ \mid\ & (\textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2) \\ \mid\ & (e'\ e_0 \ldots e_{k-1}) \\ \mid\ & (\textbf{fun } x_0 \ldots x_{k-1} \to e) \\ \mid\ & (\textbf{let } x_1 = e_1 \textbf{ in } e_0) \\ \mid\ & (\textbf{let rec } x_1 = e_1 \textbf{ and} \ldots \textbf{and } x_n = e_n \textbf{ in } e_0) \end{aligned}$$

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 **let-rec**-expression, i.e. an expression with simultaneously defined local variables.

For simplicity, we only allow `int` as basic type.

Example:

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

$$\begin{aligned} \textbf{let rec } \mathrm{fac} \quad &= \quad \textbf{fun } x \to \textbf{if } x \leq 1 \textbf{ then } 1 \\ & \qquad \textbf{else } x \cdot \mathrm{fac}\ (x - 1) \\ \textbf{in } \mathrm{fac}\ 7 & \end{aligned}$$

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

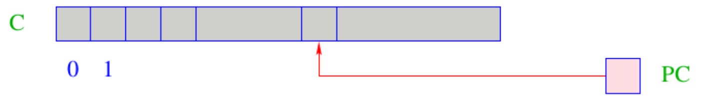
There are two Semantics:

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

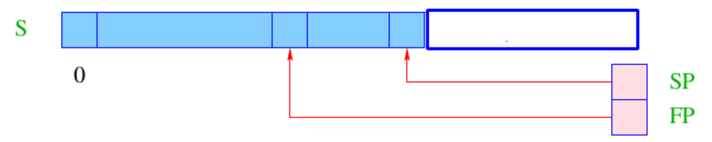
CBN: Arguments are passed unevaluated; they are only evaluated when their value is needed (as in Haskell).

12 Architecture of the MaMa:

We know already the following components:

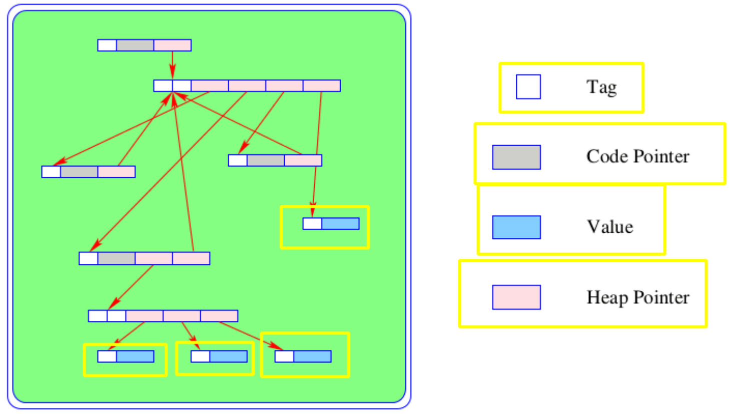


C	=	Code-store – contains the MaMa-program; each cell contains one instruction;
PC	=	Program Counter – points to the instruction to be executed next;



S	=	Runtime-Stack – each cell can hold a basic value or an address;
SP	=	Stack-Pointer – points to the topmost occupied cell;
		as in the CMa implicitely represented;
FP	=	Frame-Pointer – points to the actual stack frame.

We also need a heap H:



... it can be thought of as an abstract data type, being capable of holding data objects of the following form: