Script generated by TTT

Title: Petter: Virtual Machines (07.05.2019)

Date: Tue May 07 10:13:52 CEST 2019

Duration: 70:12 min

Pages: 11

As an exercise translate $s_1 \equiv b = (\&a) + 2;$ and $s_2 \equiv *(b+3)[0] = 5;$

$$
\begin{array}{lll}
\text{code}_R\ (e_1 \pm e_2)\ \rho & = & \text{code}_R\ e_1\ \rho \\
 & & \text{code}_R\ e_2\ \rho \\
 & & \text{loadc}\ |t| \\
 & & \text{mul} \\
 & & \text{add / sub} \quad \text{if } e_1 \text{ has type } t* \text{ or } t[]
\end{array}
$$

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

9.2 Determining Address Environments

We distinguish two kinds of variables:

1. global/extern that are defined outside of functions;
2. local/intern/automatic (inkluding formal parameters) which are defined inside functions.

$\Longrightarrow$

The address environment ρ maps names onto pairs $(tag, a) \in \{G, L\} \times \mathbb{Z}$.

Caveat

- In general, there are further refined grades of visibility of variables.
- Different parts of a program may be translated relative to different address environments!

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.

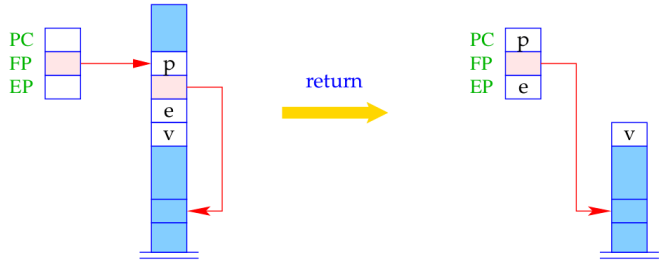
Caveat

The space requirements of the actual parameters is only known to the caller ...

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 ...

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];

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

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

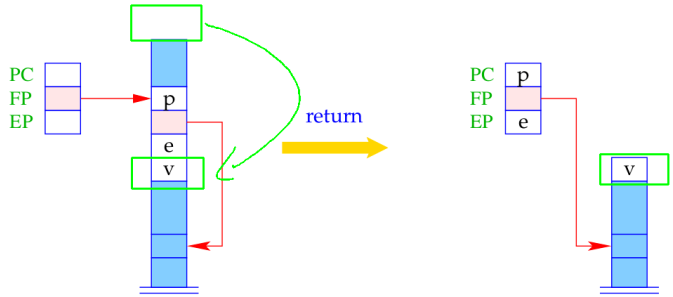
Example

For function

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

we generate:

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];

Then we define:

$$
\begin{array}{lll}
\text{code } p\ \emptyset & = & \text{enter } (\mathsf{k}+4) \\
 & & \text{alloc } (\mathsf{k}+1) \\
 & & \text{mark} \\
 & & \text{loadc \_main} \\
 & & \text{call} \\
 & & \text{slide k} \\
 & & \text{halt} \\
\_f_1\text{:} & & \text{code } F\_def_1\ \rho_1 \\
 & & \vdots \\
\_f_n\text{:} & & \text{code } F\_def_n\ \rho_n
\end{array}
$$

where $\emptyset$ $\widehat{=}$ empty address environment;

ρ_j $\widehat{=}$ global address environment before definition of f_j;

k $\widehat{=}$ size of the global variables

A program is an expression e of the form:

$$
\begin{array}{rcl}
e & ::= & b \mid x \mid (\square_1\ e) \mid (e_1\ \square_2\ e_2) \\
 & \mid & (\textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2) \\
 & \mid & (e'\ e_0 \ldots e_{k-1}) \\
 & \mid & (\textbf{fun } x_0 \ldots x_{k-1} \to e) \\
 & \mid & (\textbf{let } x_1 = e_1 \textbf{ in } e_0) \\
 & \mid & (\textbf{let rec } x_1 = e_1 \textbf{ and} \ldots \textbf{and } x_n = e_n \textbf{ in } e_0)
\end{array}
$$

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:

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

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).

Example

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

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

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).