Script generated by TTT

Title: Petter: Virtual Machines (13.05.2019)

Date: Mon May 13 10:14:35 CEST 2019

Duration: 86:53 min

Pages: 23

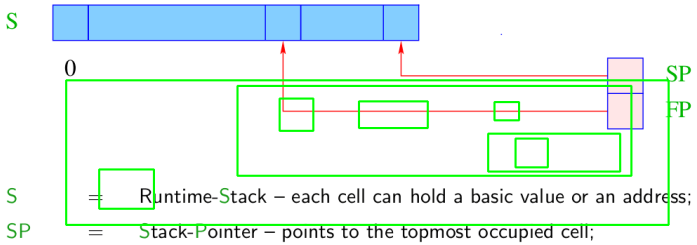
A program is an expression e of the form:

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

An expression is therefore

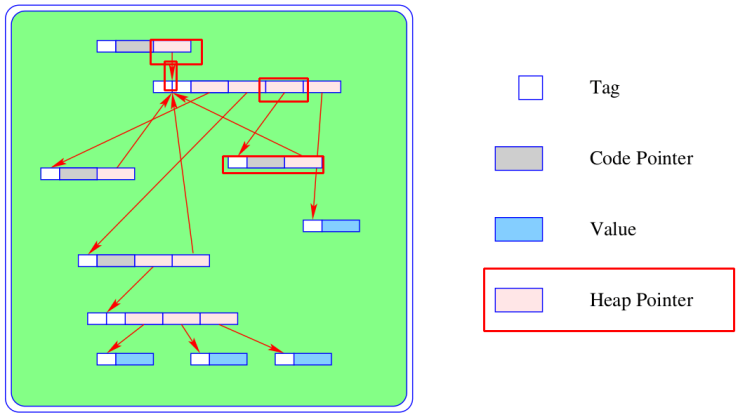
- a basic value, a variable, the application of an operator, or
- a function-application, a function-abstraction, or
- a **let**-expression, i.e. an expression with locally defined variables, or
- a **let-rec**-expression, i.e. an expression with simultaneously defined local variables.

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

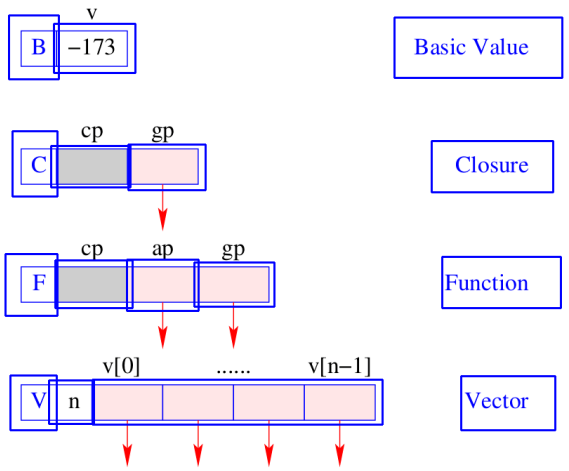


S = Runtime-Stack – each cell can hold a basic value or an address;
SP = Stack-Pointer – points to the topmost occupied cell;
as in the CMa implicitely represented;
FP = Frame-Pointer – points to the actual stack frame.

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:



The instruction new $(tag, args)$ creates a corresponding object (B, C, F, V) in H and returns a reference to it.

We distinguish three different kinds of code for an expression e:

- $\text{code}_V\ e$ — (generates code that) computes the Value of e, stores it in the heap and returns a reference to it on top of the stack (the normal case);
- $\text{code}_B\ e$ — computes the value of e, and returns it on the top of the stack (only for Basic types);
- $\text{code}_C\ e$ — does not evaluate e, but stores a Closure of e in the heap and returns a reference to the closure on top of the stack.

We start with the code schemata for the first two kinds:

13 Simple expressions

Expressions consisting only of constants, operator applications, and conditionals are translated like expressions in imperative languages:

$$
\begin{array}{lcll}
\text{code}_B\ b\ \rho\ \text{sd} & = & & \text{loadc b} \\
\text{code}_B\ (\square_1\ e)\ \rho\ \text{sd} & = & & \text{code}_B\ e\ \rho\ \text{sd};\ \text{op}_1 \\
\text{code}_B\ (e_1\ \square_2\ e_2)\ \rho\ \text{sd} & = & & \text{code}_B\ e_1\ \rho\ \text{sd} \\
 & & & \text{code}_B\ e_2\ \rho\ (\text{sd}+1) \\
 & & & \text{op}_2 \\
\text{code}_B\ (\textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2)\ \rho\ \text{sd} & = & & \text{code}_B\ e_0\ \rho\ \text{sd};\ \text{jumpz A} \\
 & & & \text{code}_B\ e_1\ \rho\ \text{sd};\ \text{jump B} \\
 & & \text{A:} & \text{code}_B\ e_2\ \rho\ \text{sd} \\
 & & \text{B:} & \ldots
\end{array}
$$

Remark

- ρ denotes the actual address environment, in which the expression is translated.
- The extra argument sd, the stack distance, *simulates* the movement of the SP when instruction execution modifies the stack. It is needed later to address variables.
- The instructions op_1 and op_2 implement the operators $\square_1$ and $\square_2$, in the same way as the the operators neg and add implement negation resp. addition in the CMa.
- For all other expressions, we first compute the value in the heap and then dereference the returned pointer:

$$
\begin{array}{lcl}
\text{code}_B\ e\ \rho\ \text{sd} & = & \text{code}_V\ e\ \rho\ \text{sd} \\
 & & \text{getbasic}
\end{array}
$$

13 Simple expressions

Expressions consisting only of constants, operator applications, and conditionals are translated like expressions in imperative languages:

$$
\begin{array}{lcll}
\text{code}_B\ b\ \rho\ \text{sd} & = & & \text{loadc b} \\
\text{code}_B\ (\square_1\ e)\ \rho\ \text{sd} & = & & \text{code}_B\ e\ \rho\ \text{sd; op}_1 \\
\text{code}_B\ (e_1\ \square_2\ e_2)\ \rho\ \text{sd} & = & & \text{code}_B\ e_1\ \rho\ \text{sd} \\
 & & & \text{code}_B\ e_2\ \rho\ (\text{sd} + 1) \\
 & & & \text{op}_2 \\
\text{code}_B\ (\textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2)\ \rho\ \text{sd} & = & & \text{code}_B\ e_0\ \rho\ \text{sd; jumpz A} \\
 & & & \text{code}_B\ e_1\ \rho\ \text{sd; jump B} \\
 & & \text{A:} & \text{code}_B\ e_2\ \rho\ \text{sd} \\
 & & \text{B:} & \ldots
\end{array}
$$

14 Accessing Variables

We must distinguish between local and global variables.

Example Regard the function f :

$$
\begin{array}{lll}
\textbf{let} & c = 5 & \\
\textbf{in let} & f = \textbf{fun}\ a & \rightarrow\ \textbf{let}\ b = a * a \\
 & & \textbf{in}\ b + c \\
\textbf{in} & f\ c &
\end{array}
$$

The function f uses the global variable c and the local variables a (as formal parameter) and b (introduced by the inner **let**).

The binding of a global variable is determined, when the function is constructed (static binding!), and later only looked up.

Accessing Global Variables

- The bindings of global variables of an expression or a function are kept in a vector in the heap (Global Vector).
- They are addressed consecutively starting with 0.
- When an F-object or a C-object are constructed, the Global Vector for the function or the expression is determined and a reference to it is stored in the gp-component of the object.
- During the evaluation of an expression, the (new) register GP (Global Pointer) points to the actual Global Vector.
- In constrast, local variables should be maintained on the stack ...

$\Longrightarrow$ General form of the address environment:

$$\rho : Vars \rightarrow \{L, G\} \times \mathbb{Z}$$

Accessing Local Variables

Local variables are administered on the stack, in stack frames.

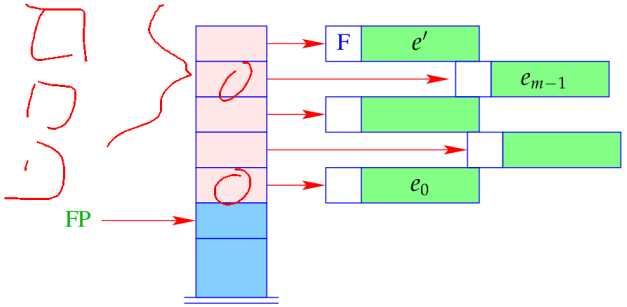
Let $e \equiv e'\ e_0\ \ldots\ e_{m-1}$ be the application of a function e' to arguments $e_0, \ldots, e_{m-1}$.

Caveat

The arity of e' does not need to be m.

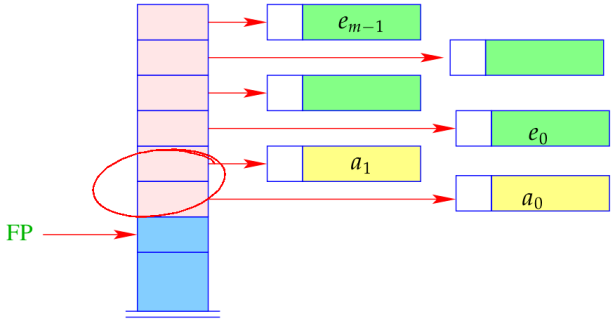
- f may therefore receive less than n arguments (under supply);
- f may also receive more than n arguments, if t is a functional type (over supply).

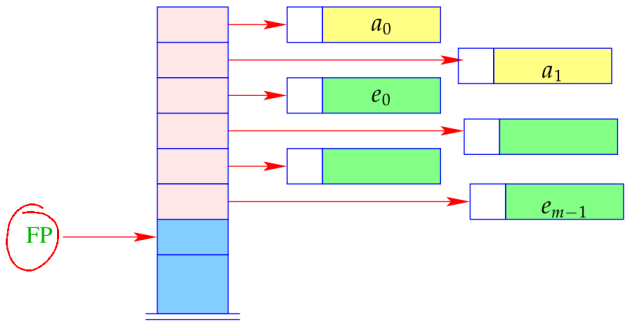
Possible stack organisations



+ Addressing of the arguments can be done relative to FP
– The local variables of e' cannot be addressed relative to FP.
– If e' is an n-ary function with $n < m$, i.e., we have an over-supplied function application, the remaining $m - n$ arguments will have to be shifted.

– If e' evaluates to a function, which has already been partially applied to the parameters $a_0, \ldots, a_{k-1}$, these have to be sneaked in underneath e_0:

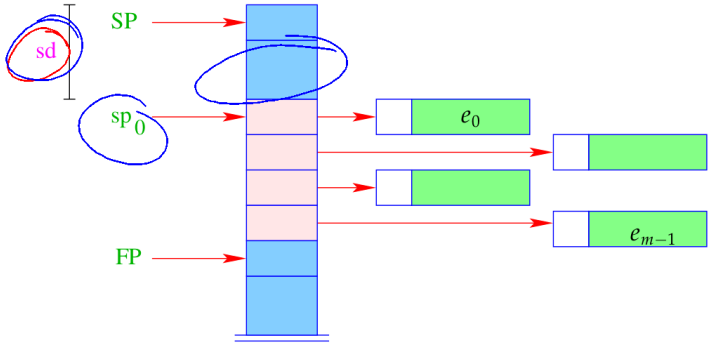




– Addressing of arguments and local variables relative to FP is no more possible. (Remember: m is unknown when the function definition is translated.)

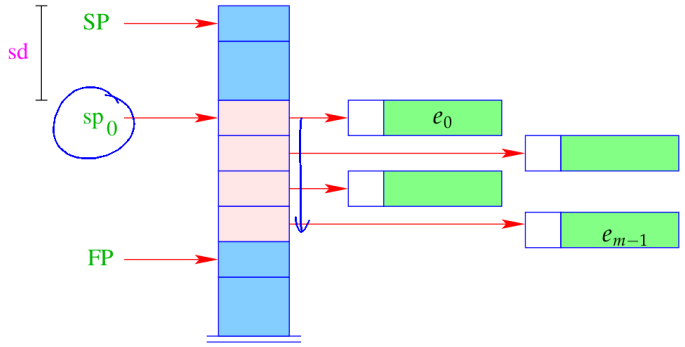
Way out

- We address both, arguments and local variables, relative to the stack pointer SP !!!
- However, the stack pointer changes during program execution...



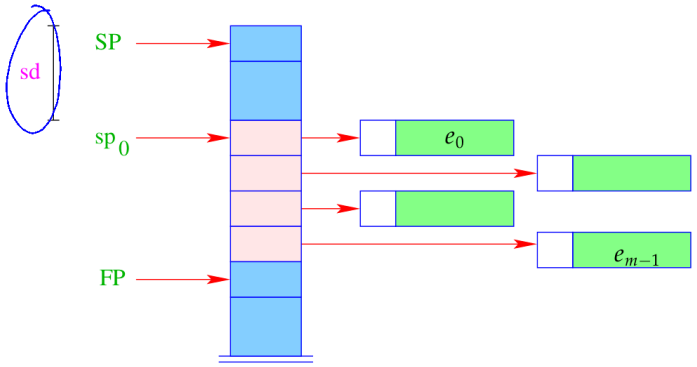
Way out

- We address both, arguments and local variables, relative to the stack pointer SP !!!
- However, the stack pointer changes during program execution...



Way out

- We address both, arguments and local variables, relative to the stack pointer SP !!!
- However, the stack pointer changes during program execution...

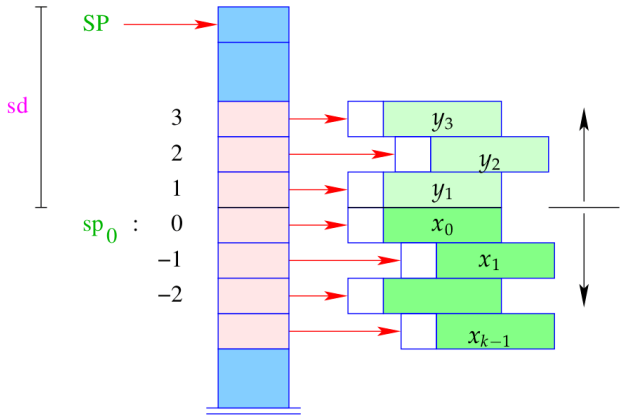


121

- The differerence between the current value of SP and its value sp_0 at the entry of the function body is called the stack distance, sd.
- Fortunately, this stack distance can be determined at compile time for each program point, by simulating the movement of the SP.
- The formal parameters $x_0, x_1, x_2, \ldots$ successively receive the non-positive relative addresses $0, -1, -2, \ldots$, i.e., $\rho\, x_i = (L, -i)$.
- The absolute address of the i-th formal parameter consequently is

$$sp_0 - i = (SP - sd) - i$$

- The local **let**-variables $y_1, y_2, y_3, \ldots$ will be successively pushed onto the stack:



- The y_i have positive relative addresses $1, 2, 3, \ldots$, that is: $\rho\, y_i = (L, i)$.
- The absolute address of y_i is then $sp_0 + i = (SP - sd) + i$

Example

Regard $e \equiv (b + c)$ for $\rho = \{b \mapsto (L,1), c \mapsto (G,0)\}$ and sd $= 1$.

With CBN, we obtain:

$\text{code}_V\ e\ \rho\ 1$	$=$	getvar $b\ \rho\ 1$	$=$	1 pushloc 0
		eval		2 eval
		getbasic		2 getbasic
		getvar $c\ \rho\ 2$		2 pushglob 0
		eval		3 eval
		getbasic		3 getbasic
		add		3 add
		mkbasic		2 mkbasic

15 let-Expressions

As a warm-up let us first consider the treatment of local variables.

Let $e \equiv \textbf{let}\ y_1 = e_1\ \textbf{in} \ldots \textbf{let}\ y_n = e_n\ \textbf{in}\ e_0$ be a nested **let**-expression.

The translation of e must deliver an instruction sequence that

- allocates local variables $y_1, \ldots, y_n$;
- in the case of
  CBV: evaluates $e_1, \ldots, e_n$ and binds the y_i to their values;
  CBN: constructs closures for the $e_1, \ldots, e_n$ and binds the y_i to them;
- evaluates the expression e_0 and returns its value.

Here, we consider the non-recursive case only, i.e. where y_j only depends on $y_1, \ldots, y_{j-1}$. We obtain for CBN:

Example

Consider the expression

$$e \equiv \textbf{let}\ a = 19\ \textbf{in let}\ b = a * a\ \textbf{in}\ a + b$$

for $\rho = \emptyset$ and sd $= 0$. We obtain (for CBV):

0	loadc 19	3	getbasic	3 pushloc 1
1	mkbasic	3	mul	4 getbasic
1	pushloc 0	2	mkbasic	4 add
2	getbasic	2	pushloc 1	3 mkbasic
2	pushloc 1	3	getbasic	3 slide 2

Example

Consider the expression

$$e \equiv \textbf{let}\ a = 19\ \textbf{in let}\ b = a * a\ \textbf{in}\ a + b$$

for $\rho = \emptyset$ and sd $= 0$. We obtain (for CBV):

0	loadc 19	3	getbasic	3 pushloc 1
1	mkbasic	3	mul	4 getbasic
1	pushloc 0	2	mkbasic	4 add
2	getbasic	2	pushloc 1	3 mkbasic
2	pushloc 1	3	getbasic	3 slide 2