Script **generated by TTT**

Title: Seidl: Virtual_Machines (30.04.2013)

Date: Tue Apr 30 14:01:29 CEST 2013

Duration: 90:36 min

Pages: 34

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 $\rho_{fac} : x \mapsto (L, -3)$ and $q = 1 + 5 = 6$.

```
_fac:  enter q     loadc 1     A: 1  loadr -3    1        mul
       alloc 0     storer -3         loadr -3    2        storer -3
       loadr -3    return            loadc 1     3        return
       loadc 0     jump B            sub         2    B:  return
       leq                           mark        4
       jumpz A                       loadc _fac  5
                                     call        5
                                     slide 0
```

where $\rho_{fac} : x \mapsto (L, -3)$ and q = ~~1 +~~ 5 ~~= 6~~.

10 Translation of Whole Programs

Before program execution, we have:

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

Let $p \equiv V\_defs \;\; F\_def_1 \ldots F\_def_n$, 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 von main();
- the instruction halt.

Then we define:

$$
\begin{array}{rrl}
\text{code } p\ \emptyset & = & \text{enter } (k+5) \\
 & & \text{alloc } (k+1) \\
 & & \text{mark} \\
 & & \text{loadc \_main} \\
 & & \text{call} \\
 & & \text{slide } (k+1) \\
 & & \text{halt} \\
 & \_f_1: & \text{code } F\_def_1\ \rho \\
 & & \vdots \\
 & \_f_n: & \text{code } F\_def_n\ \rho
\end{array}
$$

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:

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

Let $p \equiv V\_defs \;\; F\_def_1 \ldots F\_def_n$, 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 von main();
- the instruction halt.

98

Then we define:

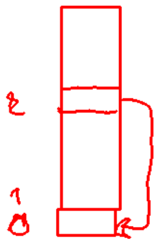
$$
\begin{array}{rrl}
\text{code } p\ \emptyset & = & \text{enter } (k+5) \\
 & & \text{alloc } (k+1) \\
 & & \text{mark} \\
 & & \text{loadc \_main} \\
 & & \text{call} \\
 & & \text{slide } (k+1) \\
 & & \text{halt} \\
 & \_f_1: & \text{code } F\_def_1\ \rho \\
 & & \vdots \\
 & \_f_n: & \text{code } F\_def_n\ \rho
\end{array}
$$

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

99

k
k+2
k+3
k+3

Then we define:



code $p\ \emptyset$ =		enter (k + 4)
		alloc (k + 1)
		mark
		loadc _main
		call
		slide (k + 1)
		halt
	_f_1:	code $F\_def_1\ \rho$
		⋮
	_f_n:	code $F\_def_n\ \rho$

where $\emptyset$ ≙ empty address environment;
ρ ≙ global address environment;
k ≙ size of the global variables

10 Translation of Whole Programs

Before program execution, we have:

SP = −1 FP = EP = 0 PC = 0 NP = MAX

Let $p \equiv V\_defs\ \ F\_def_1 \ldots F\_def_n$, 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 von main();
- the instruction halt.

Then we define:

code $p\ \emptyset$ =		enter (k + 5)
		alloc (k + 1)
		mark
		loadc _main
		call
		slide (k + 1)
		halt
	_f_1:	code $F\_def_1\ \rho$
		⋮
	_f_n:	code $F\_def_n\ \rho$

where $\emptyset$ ≙ empty address environment;
ρ ≙ global address environment;
k ≙ size of the global variables

Then we define:

code $p\ \emptyset$ =		enter (k + 5)
		alloc (k + 1)
		mark
		loadc _main
		call
		slide (k + 1)
		halt
	_f_1:	code $F\_def_1\ \rho$
		⋮
	_f_n:	code $F\_def_n\ \rho$

where $\emptyset$ ≙ empty address environment;
ρ ≙ global address environment;
k ≙ 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.

A program is an expression e of the form:

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

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:

$$
\begin{array}{rcl}
\mathbf{let\ rec}\ \mathrm{fac} & = & \mathbf{fun}\ x \to \mathbf{if}\ x \leq 1\ \mathbf{then}\ 1 \\
& & \mathbf{else}\ x \cdot \mathrm{fac}\ (x-1) \\
\mathbf{in}\ \mathrm{fac}\ 7 & &
\end{array}
$$

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:

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

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). Miranda

Example:

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

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

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

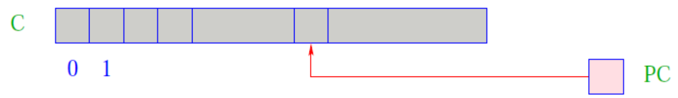
There are two Semantics:

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

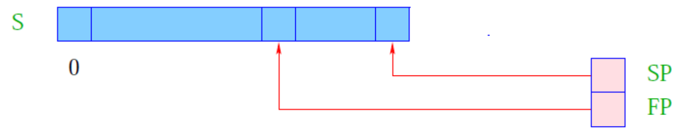
CBN: Arguments are passed unevaluated; they are only evaluated when their value is needed (as in Haskell).

12 Architecture of the MaMa:

We know already the following components:

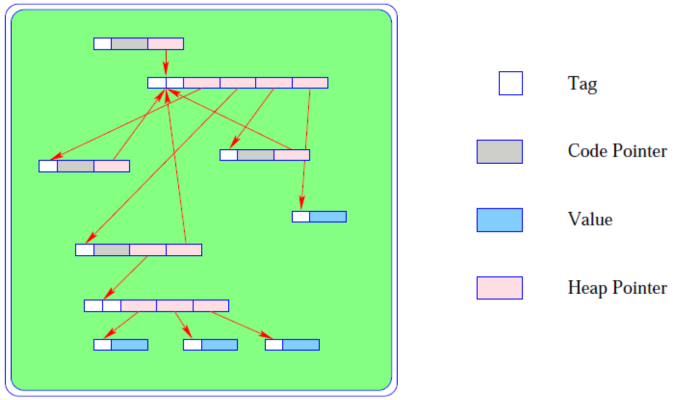


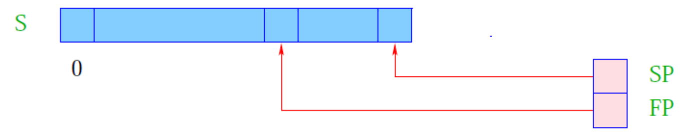
C	=	Code-store – contains the MaMa-program; each cell contains one instruction;
PC	=	Program Counter – points to the instruction to be executed next;



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.

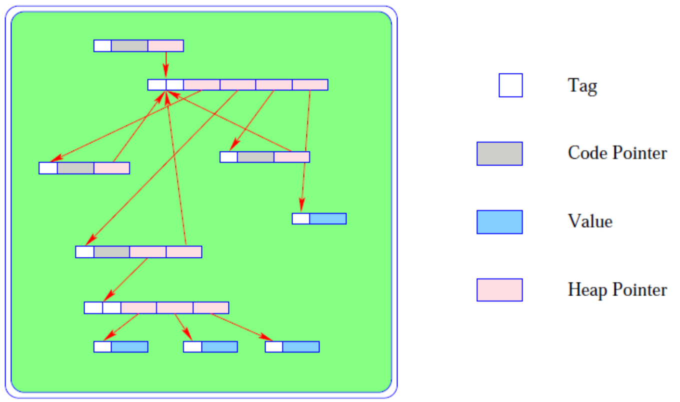
We also need a heap H:



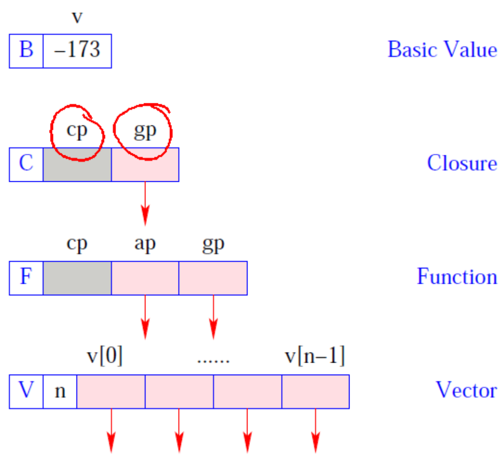


S	=	Runtime-Stack – each cell can hold a basic value or an address;
SP	=	Stack-Pointer – points to the topmost occupied cell; as in the CMa implicitely represented;
FP	=	Frame-Pointer – points to the actual stack frame.

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:

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:

108

13 Simple expressions

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

$$
\begin{array}{lcl}
\text{code}_B\ b\ \rho\ \text{sd} & = & \text{loadc b} \\
\text{code}_B\ (\Box_1\ e)\ \rho\ \text{sd} & = & \text{code}_B\ e\ \rho\ \text{sd} \\
 & & \text{op}_1 \\
\text{code}_B\ (e_1\ \Box_2\ e_2)\ \rho\ \text{sd} & = & \text{code}_B\ e_1\ \rho\ \text{sd} \\
 & & \text{code}_B\ e_2\ \rho\ (\text{sd}+1) \\
 & & \text{op}_2
\end{array}
$$

$$
\begin{array}{lcll}
\text{code}_B\ (\textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2)\ \rho\ \text{sd} & = & & \text{code}_B\ e_0\ \rho\ \text{sd} \\
 & & & \text{jumpz A} \\
 & & & \text{code}_B\ e_1\ \rho\ \text{sd} \\
 & & & \text{jump B} \\
 & & \text{A:} & \text{code}_B\ e_2\ \rho\ \text{sd} \\
 & & \text{B:} & \dots
\end{array}
$$

Note:

- ρ denotes the actual address environment, in which the expression is translated.
- The extra argument sd, the stack difference, *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 $\Box_1$ and $\Box_2$, 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:

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

13 Simple expressions

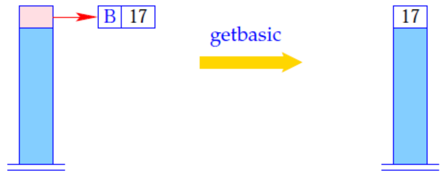
Expressions consisting only of constants, operator applications, and conditionals are translated like expressions in imperative languages:

$$\begin{aligned} \text{code}_B\ b\ \rho\ \text{sd} \quad &= \quad \text{loadc b} \\ \text{code}_B\ (\Box_1\ e)\ \rho\ \text{sd} \quad &= \quad \text{code}_B\ e\ \rho\ \text{sd} \\ & \qquad \text{op}_1 \\ \text{code}_B\ (e_1\ \Box_2\ e_2)\ \rho\ \text{sd} \quad &= \quad \text{code}_B\ e_1\ \rho\ \text{sd} \\ & \qquad \text{code}_B\ e_2\ \rho\ (\text{sd}+1) \\ & \qquad \text{op}_2 \end{aligned}$$

Note:

- ρ denotes the actual address environment, in which the expression is translated.
- The extra argument sd, the stack difference, *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 $\Box_1$ and $\Box_2$, 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:

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




```
if (H[S[SP]] != (B,_))
     Error "not basic!";
else
     S[SP] = H[S[SP]].v;
```

For code_V and simple expressions, we define analogously:

$$
\begin{array}{llll}
\text{code}_V\ b\ \rho\ \text{sd} & = & & \text{loadc b; mkbasic} \\
\text{code}_V\ (\Box_1\ e)\ \rho\ \text{sd} & = & & \text{code}_B\ e\ \rho\ \text{sd} \\
 & & & \text{op}_1\text{; mkbasic} \\
\text{code}_V\ (e_1\ \Box_2\ e_2)\ \rho\ \text{sd} & = & & \text{code}_B\ e_1\ \rho\ \text{sd} \\
 & & & \text{code}_B\ e_2\ \rho\ (\text{sd}+1) \\
 & & & \text{op}_2\text{; mkbasic} \\
\text{code}_V\ (\textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2)\ \rho\ \text{sd} & = & & \text{code}_B\ e_0\ \rho\ \text{sd} \\
 & & & \text{jumpz A} \\
 & & & \text{code}_V\ e_1\ \rho\ \text{sd} \\
 & & & \text{jump B} \\
 & & \text{A:} & \text{code}_V\ e_2\ \rho\ \text{sd} \\
 & & \text{B:} & \ldots
\end{array}
$$

For code_V and simple expressions, we define analogously:

$$
\begin{array}{llll}
\text{code}_V\ b\ \rho\ \text{sd} & = & & \text{loadc b; mkbasic} \\
\text{code}_V\ (\Box_1\ e)\ \rho\ \text{sd} & = & & \text{code}_B\ e\ \rho\ \text{sd} \\
 & & & \text{op}_1\text{; mkbasic} \\
\text{code}_V\ (e_1\ \Box_2\ e_2)\ \rho\ \text{sd} & = & & \text{code}_B\ e_1\ \rho\ \text{sd} \\
 & & & \text{code}_B\ e_2\ \rho\ (\text{sd}+1) \\
 & & & \text{op}_2\text{; mkbasic} \\
\text{code}_V\ (\textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2)\ \rho\ \text{sd} & = & & \text{code}_B\ e_0\ \rho\ \text{sd} \\
 & & & \text{jumpz A} \\
 & & & \text{code}_V\ e_1\ \rho\ \text{sd} \\
 & & & \text{jump B} \\
 & & \text{A:} & \text{code}_V\ e_2\ \rho\ \text{sd} \\
 & & \text{B:} & \ldots
\end{array}
$$

113

14 Accessing Variables

We must distinguish between local and global variables.

Example: Regard the function f :

$$
\begin{array}{ll}
\textbf{let} & c = 5 \\
\textbf{in let} & f = \textbf{fun}\ a \to \textbf{let}\ b = a * a \\
 & \qquad\qquad\quad\ \textbf{in}\ b + c \\
\textbf{in} & f\ c
\end{array}
$$

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 scoping!), and later only looked up.