Technische Universität München | *Summer Semester 08*
Fakultät für Informatik | *K N Verma*
Prof. Dr. H. Seidl | *verma@in.tum.de*

Virtual Machines

Exercise Sheet 7

Deadline: 10 June 2008, during lecture, by email, or in room 02.07.041

Exercise 1:

10 Points

Have a look at the code generated for the expression $e \equiv (a + a)$ with $\rho = \{a \mapsto (L, 1)\}$ and $sd = 1$. It was created using the Call by Need strategy.

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

The eval instructions check whether the value of a has been computed. If not, a still has to be evaluated. The second occurrence of eval in the above code is redundant, because the value of a is already known at this point.

The code generation functions can be modified such that redundant eval instructions are not generated any more. To do so, extend the code generation function for an expression e with an additional argument A. A collects the set of visible variables that are bound outside e and that have always been evaluated when reaching the code to be generated for e.

Thus the code generation scheme for variable access shall look as follows:

$$\text{code}_V\ x\ \rho\ sd\ A = \begin{cases} \text{getvar}\ x\ \rho\ sd & \text{, if } x \in A \\ \\ \text{getvar}\ x\ \rho\ sd \\ \text{eval} & \text{, otherwise} \end{cases}$$

For example:

$$\begin{array}{rcl} \text{code}_V\ (e_1\ \Box_2\ e_2)\ \rho\ \text{sd}\ A & = & \text{code}_B\ e_1\ \rho\ \text{sd}\ A \\ & & \text{code}_B\ e_2\ \rho\ (\text{sd}+1)\ A \cup A[e_1] \\ & & \text{op}_2;\ \text{mkbasic} \end{array}$$

where $A[e_1]$ is the set of free variables in the expression e_1 which already must have been evaluated in order to evaluate e_1.

a) Define formally $A[e]$, where e is a PuF expression.

b) Modify the code generation functions for PuF expressions in order to get rid of redundant eval instructions.

Exercise 2:

10 Points

Extend PuF with type Tree. Trees are constructed using the nullary constructor (constant) LEAF and the 3-ary constructor NODE. NODE constructs a Tree value from an arbitrary value and two Tree values. The syntax of expressions e is extended with:

$$\begin{array}{rcl} e & ::= & \ldots \mid LEAF \mid NODE(e_1, e_2, e_3) \\ & & \mid (\textbf{case}\ e_0\ \textbf{of}\ LEAF \to e_1;\ \ NODE(info, left, right) \to e_2) \end{array}$$

Define code generation functions for the new expressions. Extend the set of heap objects with new objects of type Tree. You may define new MaMa instructions.