Technische Universität München *Summer 2011*
Fakultät für Informatik *Kalmer Apinis*
Prof. Dr. H. Seidl *apinis@in.tum.de*

Virtual Machines

Exercise Sheet 7

Deadline: July 5th, 2011, 14:00

Exercise 1: *Tail Recursion* *4 Points*

A function application $l \equiv e'\ e_1\ \ldots\ e_n$ within some arbitrary expression e_g is identified as a so-called *last call* if the evaluation of l can deliver the value of e_g. In order to perform *tail call optimization* during the translation of e_g, all occurrences of *last calls* have to be determined for e_g.

Develop a general schema to determine the occurrences of last calls in any expression.

Exercise 2: `eval` *Optimization* *8 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.

`getvar` $a\ \rho\ 1$	$\equiv$	1	`pushloc 0`
`eval`	$\equiv$	2	`eval`
`getbasic`	$\equiv$	2	`getbasic`
`getvar` $a\ \rho\ 2$	$\equiv$	2	`pushloc 1`
`eval`	$\equiv$	3	`eval`
`getbasic`	$\equiv$	3	`getbasic`
`add`	$\equiv$	3	`add`
`mkbasic`	$\equiv$	2	`mkbasic`

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

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

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

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

For example:

$$\begin{array}{lcl} \mathsf{code}_V\ (e_1\ \Box_2\ e_2)\ \rho\ sd\ A & \equiv & \mathsf{code}_B\ e_1\ \rho\ sd\ A \\ & & \mathsf{code}_B\ e_2\ \rho\ (sd+1)\ (A \cup A[e_1]) \\ & & \mathtt{op}_2 \\ & & \mathtt{mkbasic} \end{array}$$

where $A[e_1]$ is the set of free variables in the expression e_1 which had to be evaluated during the evaluation of e_1.

1. Define $A[e]$ formally, where e is an arbitrary expression.
2. Modify the code generation functions for expressions in order to get rid of redundant `eval` instructions.

Exercise 3: *Type Extensions* *8 Points*

Introduce the new type `'a Tree`. Trees are constructed using the nullary constructor (constant) `Leaf` and the ternary constructor `Node of 'a * 'a Tree * 'a Tree`. The syntax is then extended with:

$$\begin{array}{lcl} e & ::= & \ldots \mid \mathtt{Leaf} \mid \mathtt{Node}(e_1, e_2, e_3) \\ & & \mid (\mathtt{match}\ e_0\ \mathtt{with}\ \mathtt{Leaf} \rightarrow e_1 \mid \mathtt{Node}(\mathtt{info}, \mathtt{left}, \mathtt{right}) \rightarrow e_2) \end{array}$$

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