Script **generated by TTT**

Title: Petter: Virtual Machines (20.05.2019)

Date: Mon May 20 10:14:57 CEST 2019

Duration: 66:34 min

Pages: 14

20 Closures and their Evaluation

- Closures are needed in the implementation of CBN for **let**-, **let-rec** expressions as well as for actual paramaters of functions.
- Before the value of a variable is accessed (with CBN), this value must be available.
- Otherwise, a stack frame must be created to determine this value.
- This task is performed by the instruction eval.

eval can be decomposed into small actions:

```
eval  =  if (H[S[SP]] ≡ (C,_,_)) {
             mark0;          // allocation of the stack frame
             pushloc 3;      // copying of the reference
             apply0;         // corresponds to apply
         }
```

- A closure can be understood as a parameterless function. Thus, there is no need for an ap-component.
- Evaluation of the closure means evaluation of an application of this function to 0 arguments.
- In constrast to mark A , mark0 dumps the current PC.
- The difference between apply and apply0 is that no argument vector is put onto the stack.

eval can be decomposed into small actions:

```
eval  =  if (H[S[SP]] ≡ (C,_,_)) {
             mark0;          // allocation of the stack frame
             pushloc 3;      // copying of the reference
             apply0;         // corresponds to apply
         }
```

- A closure can be understood as a parameterless function. Thus, there is no need for an ap-component.
- Evaluation of the closure means evaluation of an application of this function to 0 arguments.
- In constrast to mark A , mark0 dumps the current PC.
- The difference between apply and apply0 is that no argument vector is put onto the stack.

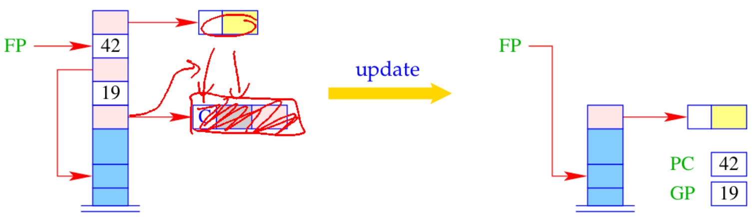
163

In fact, the instruction update is the combination of the two actions:

popenv

rewrite 1

It overwrites the closure with the computed value.



eval can be decomposed into small actions:

```
eval  =  if (H[S[SP]] ≡ (C,_,_)) {
             mark0;          // allocation of the stack frame
             pushloc 3;      // copying of the reference
             apply0;         // corresponds to apply
         }
```

- A closure can be understood as a parameterless function. Thus, there is no need for an ap-component.
- Evaluation of the closure means evaluation of an application of this function to 0 arguments.
- In constrast to mark A , mark0 dumps the current PC.
- The difference between apply and apply0 is that no argument vector is put onto the stack.

23 The Translation of a Program Expression

Execution of a program e starts with

$$\mathsf{PC} = 0 \qquad \mathsf{SP} = \mathsf{FP} = \mathsf{GP} = -1$$

The expression e must not contain free variables.

The value of e should be determined and then a halt instruction should be executed.

$$\mathsf{code}\ e \;=\; \mathsf{code}_V\ e\ \emptyset\ 0$$
$$\mathsf{halt}$$

Remarks

- The code schemata as defined so far produce Spaghetti code.
- Reason: Code for function bodies and closures placed directly behind the instructions mkfunval resp. mkclos with a jump over this code.
- Alternative: Place this code somewhere else, e.g. following the halt-instruction:

  Advantage: Elimination of the direct jumps following mkfunval and mkclos.

  Disadvantage: The code schemata are more complex as they would have to accumulate the code pieces in a Code-Dump.

$\Longrightarrow$

Solution

Disentangle the Spaghetti code in a subsequent optimization phase.

- In order to construct a tuple, we collect sequence of references on the stack. Then we construct a vector of these references in the heap using mkvec
- For returning components we use an indexed access into the tuple.

$$\text{code}_V\ (e_0,\ldots,e_{k-1})\ \rho\ \text{sd} = \text{code}_C\ e_0\ \rho\ \text{sd}$$
$$\text{code}_C\ e_1\ \rho\ (\text{sd}+1)$$
$$\ldots$$
$$\text{code}_C\ e_{k-1}\ \rho\ (\text{sd}+k-1)$$
mkvec k

$$\text{code}_V\ (\#j\ e)\ \rho\ \text{sd} = \text{code}_V\ e\ \rho\ \text{sd}$$
get j
eval

In the case of CBV, we directly compute the values of the e_i.

Deconstruction: Accessing all components of a tuple simulataneously:

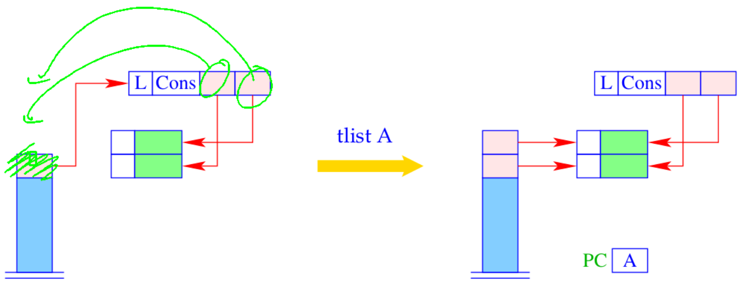
$$e \equiv \textbf{let}\ (y_0,\ldots,y_{k-1}) = e_1\ \textbf{in}\ e_0$$

This is translated as follows:

$$\text{code}_V\ e\ \rho\ \text{sd} = \text{code}_V\ e_1\ \rho\ \text{sd}$$
getvec k
$$\text{code}_V\ e_0\ \rho'\ (\text{sd}+k)$$
slide k

where $\rho' = \rho \oplus \{y_i \mapsto (L, sd+i+1) \mid i = 0,\ldots,k-1\}$.

The instruction getvec k pushes the components of a vector of length k onto the stack:




```
... else {
  S[SP+1] = S[SP]→s[1];
  S[SP] = S[SP]→s[0];
  SP++; PC = A;
}
```


Example The (disentangled) body of the function app with app $\mapsto (G,0)$:

0		targ 2	3		pushglob 0	0	C:	mark D
0		pushloc 0	4		pushloc 2	3		pushglob 2
1		eval	5		pushloc 6	4		pushglob 1
1		tlist A	6		mkvec 3	5		pushglob 0
0		pushloc 1	4		mkclos C	6		eval
1		eval	4		cons	6		apply
1		jump B	3		slide 2	1	D:	update
2	A:	pushloc 1	1	B:	return 2			

Remark

Datatypes with more than two constructors need a generalization of the tlist instruction, corresponding to a `switch`-instruction.

Example The (disentangled) body of the function app with app $\mapsto (G, 0)$:

0		targ 2	3		pushglob 0	0	C:	mark D
0		pushloc 0	4		pushloc 2	3		pushglob 2
1		eval	5		pushloc 6	4		pushglob 1
1		tlist A	6		mkvec 3	5		pushglob 0
0		pushloc 1	4		mkclos C	6		eval
1		eval	4		cons	6		apply
1		jump B	3		slide 2	1	D:	update
2	A:	pushloc 1	1	B:	return 2			

Remark

Datatypes with more than two constructors need a generalization of the tlist instruction, corresponding to a `switch`-instruction.

24.5 Closures of Tuples and Lists

The general schema for code_C can be optimized for tuples and lists:

$$\begin{array}{lllll}
\text{code}_C\ (e_0, \ldots, e_{k-1})\ \rho\ \text{sd} & = & \text{code}_V\ (e_0, \ldots, e_{k-1})\ \rho\ \text{sd} & = & \text{code}_C\ e_0\ \rho\ \text{sd} \\
 & & & & \text{code}_C\ e_1\ \rho\ (\text{sd}+1) \\
 & & & & \ldots \\
 & & & & \text{code}_C\ e_{k-1}\ \rho\ (\text{sd}+k-1) \\
 & & & & \text{mkvec k} \\
\text{code}_C\ []\ \rho\ \text{sd} & = & \text{code}_V\ []\ \rho\ \text{sd} & = & \text{nil} \\
\text{code}_C\ (e_1 :: e_2)\ \rho\ \text{sd} & = & \text{code}_V\ (e_1 :: e_2)\ \rho\ \text{sd} & = & \text{code}_C\ e_1\ \rho\ \text{sd} \\
 & & & & \text{code}_C\ e_2\ \rho\ (\text{sd}+1) \\
 & & & & \text{cons}
\end{array}$$

24.5 Closures of Tuples and Lists

The general schema for code_C can be optimized for tuples and lists:

$$\begin{array}{lllll}
\text{code}_C\ (e_0, \ldots, e_{k-1})\ \rho\ \text{sd} & = & \text{code}_V\ (e_0, \ldots, e_{k-1})\ \rho\ \text{sd} & = & \text{code}_C\ e_0\ \rho\ \text{sd} \\
 & & & & \text{code}_C\ e_1\ \rho\ (\text{sd}+1) \\
 & & & & \ldots \\
 & & & & \text{code}_C\ e_{k-1}\ \rho\ (\text{sd}+k-1) \\
 & & & & \text{mkvec k} \\
\text{code}_C\ []\ \rho\ \text{sd} & = & \text{code}_V\ []\ \rho\ \text{sd} & = & \text{nil} \\
\text{code}_C\ (e_1 :: e_2)\ \rho\ \text{sd} & = & \text{code}_V\ (e_1 :: e_2)\ \rho\ \text{sd} & = & \text{code}_C\ e_1\ \rho\ \text{sd} \\
 & & & & \text{code}_C\ e_2\ \rho\ (\text{sd}+1) \\
 & & & & \text{cons}
\end{array}$$

211