Script generated by TTT

Title: Seidl: Virtual Machines (13.05.2014)

Date: Tue May 13 10:15:09 CEST 2014

Duration: 92:17 min

Pages: 37

For CBN, we obtain:

$$
\begin{array}{lll}
\text{code}_V\ e\ \rho\ \text{sd} & = & \text{alloc n} \qquad\qquad // \text{ allocates local variables} \\
& & \text{code}_C\ e_1\ \rho'\ (\text{sd}+n) \\
& & \text{rewrite n} \\
& & \ldots \\
& & \text{code}_C\ e_n\ \rho'\ (\text{sd}+n) \\
& & \text{rewrite 1} \\
& & \text{code}_V\ e_0\ \rho'\ (\text{sd}+n) \\
& & \text{slide n} \qquad\qquad // \text{ deallocates local variables}
\end{array}
$$

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

In the case of CBV, we also use code_V for the expressions $e_1, \ldots, e_n$.

Warning:

Recursive definitions of basic values are undefined with CBV!!!

Example:

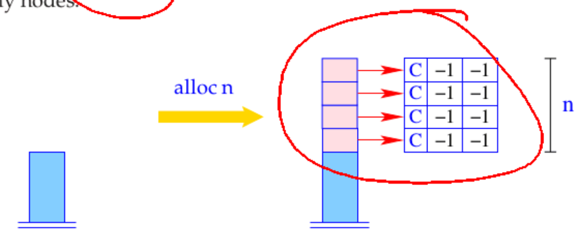
Consider the expression

$$e \equiv \textbf{let rec } f = \textbf{fun } x\ y \to \textbf{if } y \leq 1 \textbf{ then } x \textbf{ else } f\,(x*y)\,(y-1) \textbf{ in } f\ 1$$

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

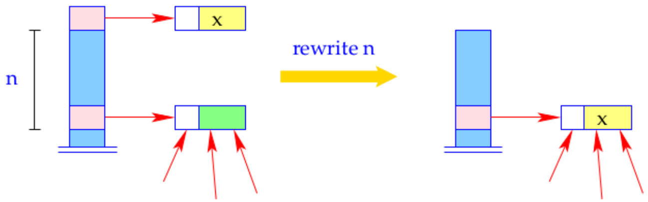
0	alloc 1	0	A:	targ 2	4		loadc 1
1	pushloc 0	0		...	5		mkbasic
2	mkvec 1	1		return 2	5		pushloc 4
2	mkfunval A	2	B:	rewrite 1	6		apply
2	jump B	1		mark C	2	C:	slide 1

The instruction alloc n reserves n cells on the stack and initialises them with n dummy nodes:



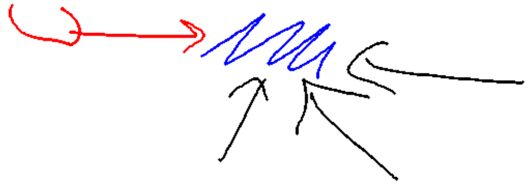

```
for (i=1; i<=n; i++)
    S[SP+i] = new (C,-1,-1);
SP = SP + n;
```

The instruction rewrite n overwrites the contents of the heap cell pointed to by the reference at S[SP–n]:



H[S[SP-n]] = H[S[SP]];
SP = SP - 1;

- The reference S[SP – n] remains unchanged!
- Only its contents is changed!



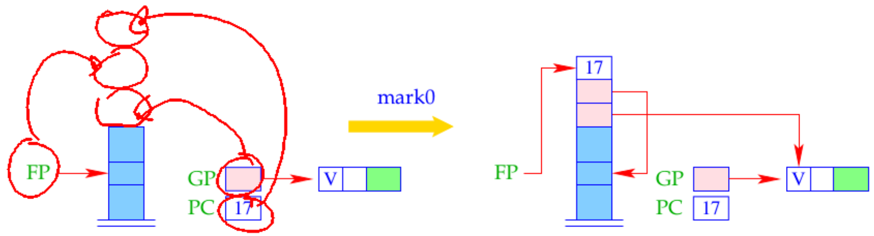
20 Closures and their Evaluation

- Closures are needed for the implementation of CBN and for functional paramaters.
- 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 thus 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 on the stack.

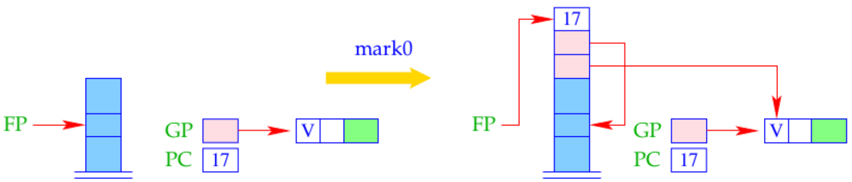


S[SP+1] = GP;
S[SP+2] = FP;
S[SP+3] = PC;
FP = SP = SP + 3;

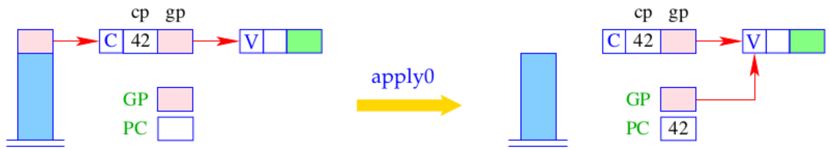
eval can be decomposed into small actions:

```
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 thus 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 on the stack.

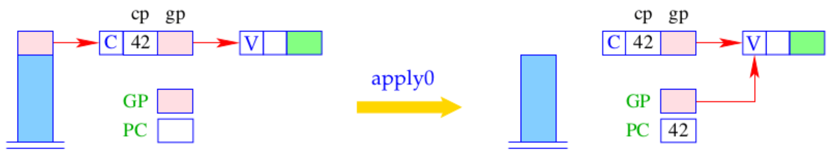


S[SP+1] = GP;
S[SP+2] = FP;
S[SP+3] = PC;
FP = SP = SP + 3;



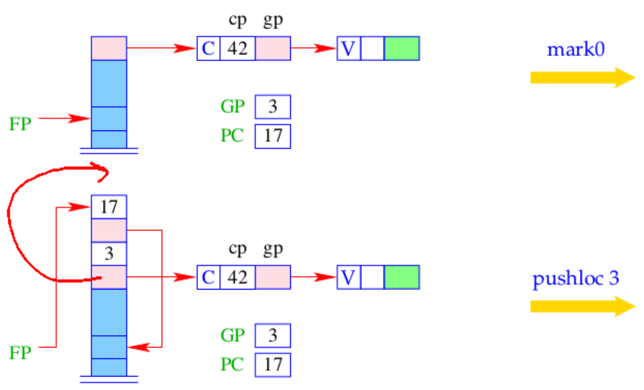
h = S[SP]; SP--;
GP = h→gp; PC = h→cp;

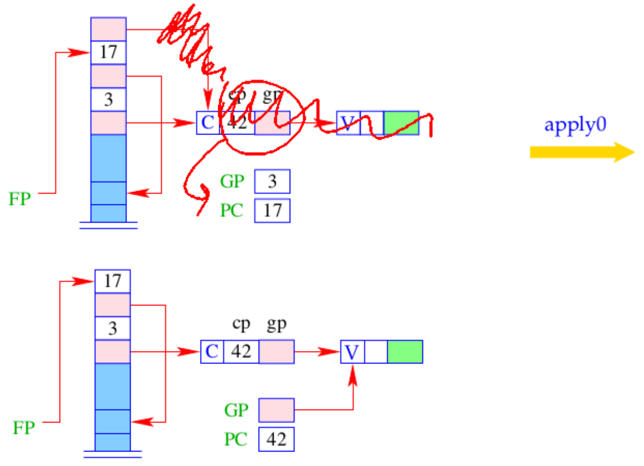
We thus obtain for the instruction eval:

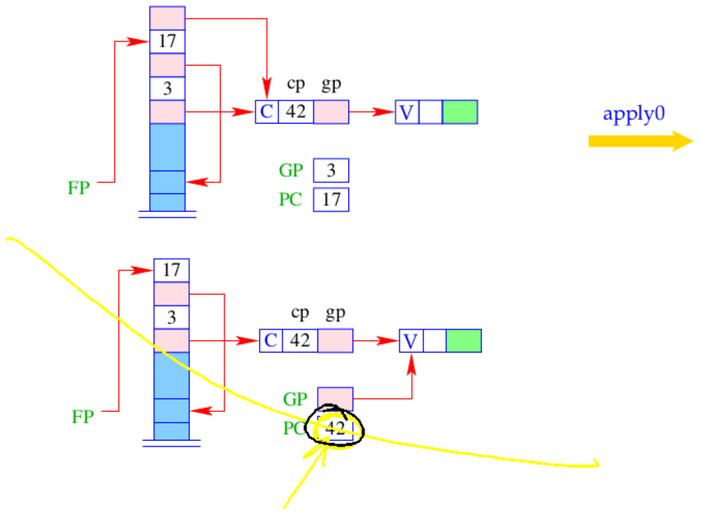


h = S[SP]; SP--;
GP = h→gp; PC = h→cp;

We thus obtain for the instruction eval:







The construction of a closure for an expression e consists of:

- Packing the bindings for the free variables into a vector;
- Creation of a C-object, which contains a reference to this vector and to the code for the evaluation of e:

$$
\begin{array}{rrl}
\text{code}_C\; e\; \rho\; \text{sd} & = & \text{getvar}\; z_0\; \rho\; \text{sd} \\
 & & \text{getvar}\; z_1\; \rho\; (\text{sd}+1) \\
 & & \ldots \\
 & & \text{getvar}\; z_{g-1}\; \rho\; (\text{sd}+g-1) \\
 & & \text{mkvec}\; g \\
 & & \text{mkclos}\; A \\
 & & \text{jump}\; B \\
 & A: & \text{code}_V\; e\; \rho'\; 0 \\
 & & \boxed{\text{update}} \\
 & B: & \ldots
\end{array}
$$

where $\{z_0, \ldots, z_{g-1}\} = \mathit{free}(e)$ and $\rho' = \{z_i \mapsto (G, i) \mid i = 0, \ldots, g-1\}$.

Example:

Consider $e \equiv a * a$ with $\rho = \{a \mapsto (L, 0)\}$ and sd $= 1$. We obtain:

1	pushloc 1	0	A:	pushglob 0	2		getbasic
2	mkvec 1	1		eval	2		mul
2	mkclos A	1		getbasic	1		mkbasic
2	jump B	1		pushglob 0	1		update
		2		eval	2	B:	...

Example:

Consider $e \equiv a * a$ with $\rho = \{a \mapsto (L, 0)\}$ and sd $= 1$. We obtain:

1	pushloc 1	0	A:	pushglob 0	2		getbasic
2	mkvec 1	1		eval	2		mul
2	mkclos A	1		getbasic	1		mkbasic
2	jump B	1		pushglob 0	1		update
		2		eval	2	B:	...

170

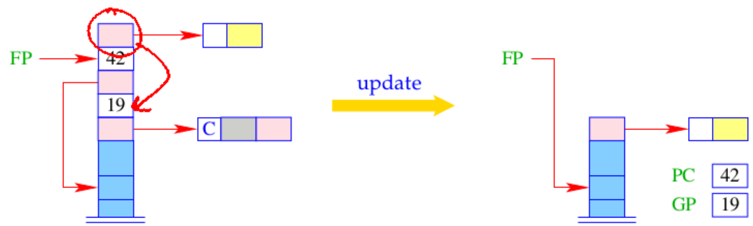
- The instruction mkclos A is analogous to the instruction mkfunval A.
- It generates a C-object, where the included code pointer is A.

S[SP] = new (C, A, S[SP]);

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

popenv
rewrite 1

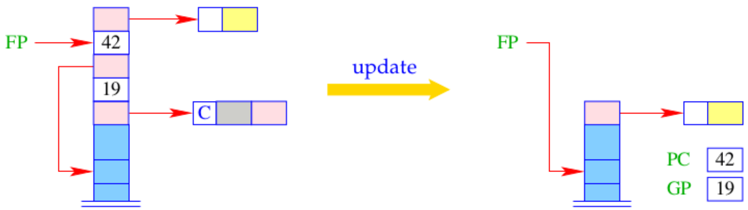
It overwrites the closure with the computed value.



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

popenv
rewrite 1

It overwrites the closure with the computed value.



21 Optimizations I: Global Variables

Observation:

- Functional programs construct many F- and C-objects.
- This requires the inclusion of (the bindings of) all global variables. Recall, e.g., the construction of a closure for an expression e ...

$$
\begin{array}{rlll}
\text{code}_C\ e\ \rho\ \text{sd} & = & & \text{getvar}\ z_0\ \rho\ \text{sd} \\
 & & & \text{getvar}\ z_1\ \rho\ (\text{sd}+1) \\
 & & & \ldots \\
 & & & \text{getvar}\ z_{g-1}\ \rho\ (\text{sd}+g-1) \\
 & & & \text{mkvec g} \\
 & & & \text{mkclos A} \\
 & & & \text{jump B} \\
 & & \text{A}: & \text{code}_V\ e\ \rho'\ 0 \\
 & & & \text{update} \\
 & & \text{B}: & \ldots
\end{array}
$$

where $\{z_0, \ldots, z_{g-1}\} = free(e)$ and $\rho' = \{z_i \mapsto (G, i) \mid i = 0, \ldots, g-1\}$.

Idea:

- Reuse Global Vectors, i.e. share Global Vectors!
- Profitable in the translation of **let**-expressions or function applications: Build one Global Vector for the union of the free-variable sets of all let-definitions resp. all arguments.
- Allocate (references to) global vectors with multiple uses in the stack frame like local variables!
- Support the access to the current GP by an instruction copyglob :

- The optimization will cause Global Vectors to contain more components than just references to the free the variables that occur in one expression ...

Disadvantage: Superfluous components in Global Vectors prevent the deallocation of already useless heap objects $\Longrightarrow$ Space Leaks :-(

Potential Remedy: Deletion of references at the end of their life time.

22 Optimizations II: Closures

In some cases, the construction of closures can be avoided, namely for

- Basic values,
- Variables,
- Functions.

Basic Values:

The construction of a closure for the value is at least as expensive as the construction of the B-object itself!

Therefore:

$$\text{code}_C\ b\ \rho\ \text{sd} \quad = \quad \text{code}_V\ b\ \rho\ \text{sd} \quad = \quad \begin{array}{l}\text{loadc b}\\ \text{mkbasic}\end{array}$$

This replaces:

mkvec 0		jump B	mkbasic	B:	...
mkclos A	A:	loadc b	update		

Variables:

Variables are either bound to values or to C-objects. Constructing another closure is therefore superfluous. Therefore:

$$\text{code}_C\ x\ \rho\ \text{sd} \quad = \quad \text{getvar}\ x\ \rho\ \text{sd}$$

This replaces:

getvar x ρ sd	mkclos A	A:	pushglob 0		update
mkvec 1	jump B		eval	B:	...

Example:

Consider $e \equiv$ **let rec** $a = b$ **and** $b = 7$ **in** a.

$\text{code}_V\; e\; \emptyset\; 0$ produces:

0	alloc 2	3	rewrite 2	3	mkbasic	2	pushloc 1
2	pushloc 0	2	loadc 7	3	rewrite 1	3	eval
						3	slide 2

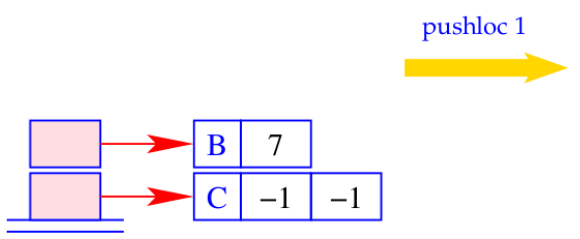
The execution of this instruction sequence should deliver the basic value 7 ...

0	alloc 2	3	rewrite 2	3	mkbasic	2	pushloc 1
2	pushloc 0	2	loadc 7	3	rewrite 1	3	eval
						3	slide 2

0	alloc 2	3	rewrite 2	3	mkbasic	2	pushloc 1
2	pushloc 0	2	loadc 7	3	rewrite 1	3	eval
						3	slide 2

Segmentation Fault !!

0	alloc 2	3	rewrite 2	3	mkbasic	2	pushloc 1
2	pushloc 0	2	loadc 7	3	rewrite 1	3	eval
						3	slide 2



0	alloc 2	3	rewrite 2	3	mkbasic	2	pushloc 1
2	pushloc 0	2	loadc 7	3	rewrite 1	3	eval
						3	slide 2

Functions:

Functions are values, which are not evaluated further. Instead of generating code that constructs a closure for an F-object, we generate code that constructs the F-object directly.

Therefore:

$$\text{code}_C\ (\textbf{fun}\ x_0 \ldots x_{k-1} \rightarrow e)\ \rho\ \text{sd} \quad = \quad \text{code}_V\ (\textbf{fun}\ x_0 \ldots x_{k-1} \rightarrow e)\ \rho\ \text{sd}$$

Functions:

Functions are values, which are not evaluated further. Instead of generating code that constructs a closure for an F-object, we generate code that constructs the F-object directly.

Therefore:

$$\text{code}_C\ (\textbf{fun}\ x_0 \ldots x_{k-1} \rightarrow e)\ \rho\ \text{sd} \quad = \quad \text{code}_V\ (\textbf{fun}\ x_0 \ldots x_{k-1} \rightarrow e)\ \rho\ \text{sd}$$

23 The Translation of a Program Expression

Execution of a program e starts with

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

The expression e must not contain free variables.

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

$$\begin{array}{lll} \text{code}\ e & = & \text{code}_V\ e\ \emptyset\ 0 \\ & & \text{halt} \end{array}$$

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 :-)

Example: **let** $a = 17$ **in let** $f =$ **fun** $b \to a + b$ **in** $f\ 42$

Disentanglement of the jumps produces:

0	loadc 17	2	mark B	3	B:	slide 2	1	pushloc 1
1	mkbasic	5	loadc 42	1		halt	2	eval
1	pushloc 0	6	mkbasic	0	A:	targ 1	2	getbasic
2	mkvec 1	6	pushloc 4	0		pushglob 0	2	add
2	mkfunval A	7	eval	1		eval	1	mkbasic
		7	apply	1		getbasic	1	return 1