Script generated by TTT

Title: Seidl: Virtual_Machines (19.06.2012)

Date: Tue Jun 19 14:01:39 CEST 2012

Duration: 80:10 min

Pages: 57

18 Over– and Undersupply of Arguments

The first instruction to be executed when entering a function body, i.e., after an apply is targ k .

This instruction checks whether there are enough arguments to evaluate the body.

Only if this is the case, the execution of the code for the body is started.

Otherwise, i.e. in the case of under-supply, a new F-object is returned.

The test for number of arguments uses: SP – FP

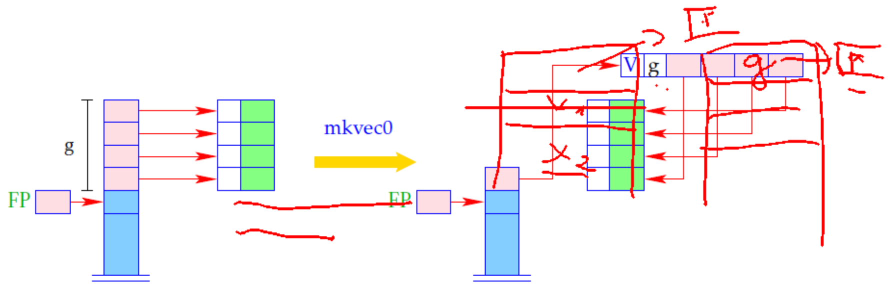
targ k is a complex instruction.

We decompose its execution in the case of under-supply into several steps:

```
targ k  =  if (SP – FP < k) {
               mkvec0;     // creating the argumentvector
               wrap;       // wrapping into an F – object
               popenv;     // popping the stack frame
           }
```

The combination of these steps into one instruction is a kind of optimization :-)

The instruction mkvec0 takes all references from the stack above FP and stores them into a vector:



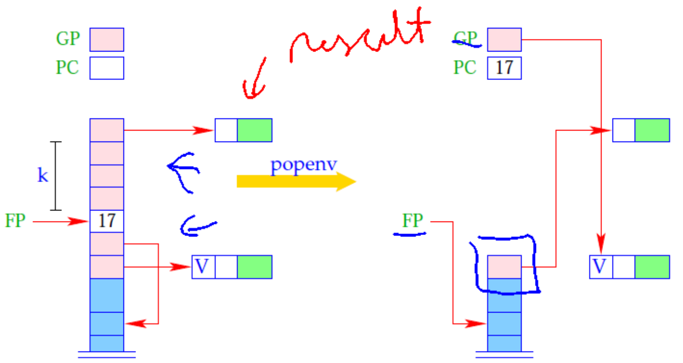

```
g = SP–FP; h = new (V, g);
SP = FP+1;
for (i=0; i<g; i++)
    h→v[i] = S[SP + i];
S[SP] = h;
```

- The stack frame can be released after the execution of the body if exactly the right number of arguments was available.
- If there is an oversupply of arguments, the body must evaluate to a function, which consumes the rest of the arguments ...
- The check for this is done by return k:

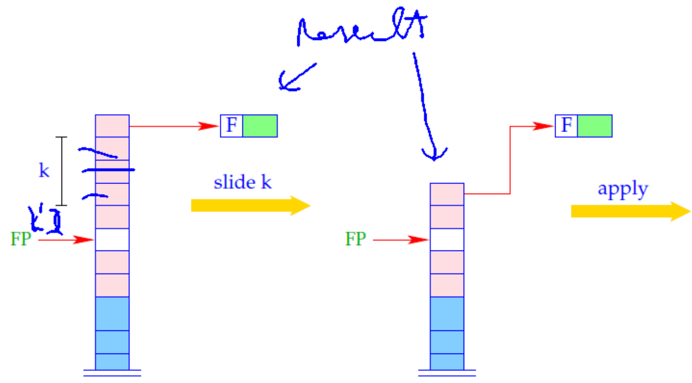
```
return k  =  if (SP − FP = k + 1)
                popenv;              // Done
             else {                  // There are more arguments
                slide k;
                apply;               // another application
             }
```

The execution of return k results in:

Case: Done



Case: Over-supply



19 let-rec-Expressions

Consider the expression $e \equiv \textbf{let rec } y_1 = e_1 \textbf{ and} \ldots \textbf{and } y_n = e_n \textbf{ in } e_0$.

The translation of e must deliver an instruction sequence that

- allocates local variables $y_1, \ldots, y_n$;
- in the case of
  CBV: evaluates $e_1, \ldots, e_n$ and binds the y_i to their values;
  CBN: constructs closures for the $e_1, \ldots, e_n$ and binds the y_i to them;
- evaluates the expression e_0 and returns its value.

Warning:

In a **letrec**-expression, the definitions can use variables that will be allocated only later! $\Longrightarrow$ Dummy-values are put onto the stack before processing the definition.

For CBN, we obtain:

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

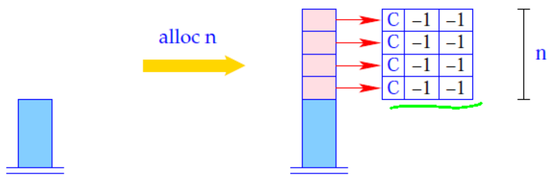
where $\rho' = \rho \oplus \{y_i \mapsto (L, \text{sd}+i) \mid i = 1, \ldots, n\}$.

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!!!

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;
```


19 let-rec-Expressions

Consider the expression $e \equiv \textbf{let rec}\ y_1 = e_1\ \textbf{and} \ldots \textbf{and}\ y_n = e_n\ \textbf{in}\ e_0$.

The translation of e must deliver an instruction sequence that

- allocates local variables $y_1, \ldots, y_n$;
- in the case of
  CBV: evaluates $e_1, \ldots, e_n$ and binds the y_i to their values;
  CBN: constructs closures for the $e_1, \ldots, e_n$ and binds the y_i to them;
- evaluates the expression e_0 and returns its value.

Warning:

In a **letrec**-expression, the definitions can use variables that will be allocated only later! $\Longrightarrow$ Dummy-values are put onto the stack before processing the definition.

For CBN, we obtain:

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

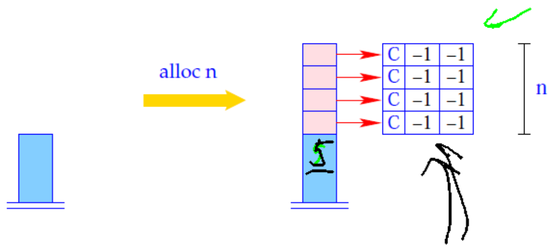
where $\rho' = \rho \oplus \{y_i \mapsto (L, \text{sd}+i) \mid i = 1, \ldots, n\}$.

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!!!

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;
```


For CBN, we obtain:

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

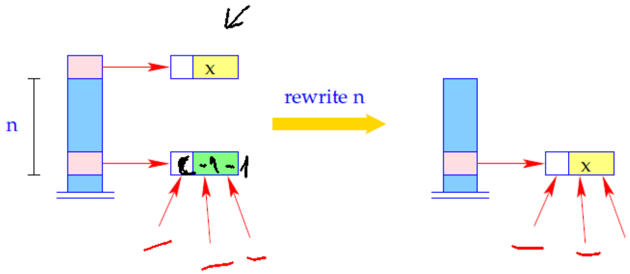
where $\rho' = \rho \oplus \{y_i \mapsto (L, \text{sd}+i) \mid i = 1, \ldots, n\}$.

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!!!

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!

For CBN, we obtain:

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

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!!!

For CBN, we obtain:

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

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!!!

For CBN, we obtain:

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

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

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

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

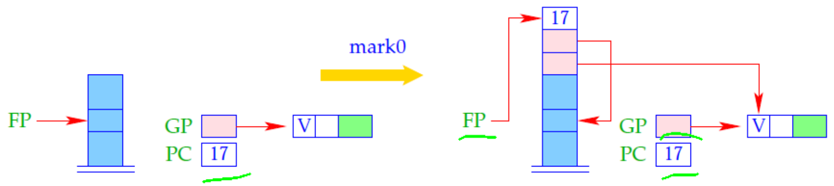
20 Closures and their Evaluation

- 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.

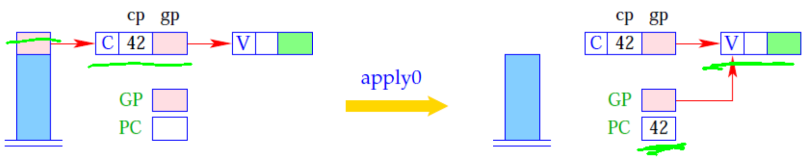
eval can be decomposed into small actions:

```
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;



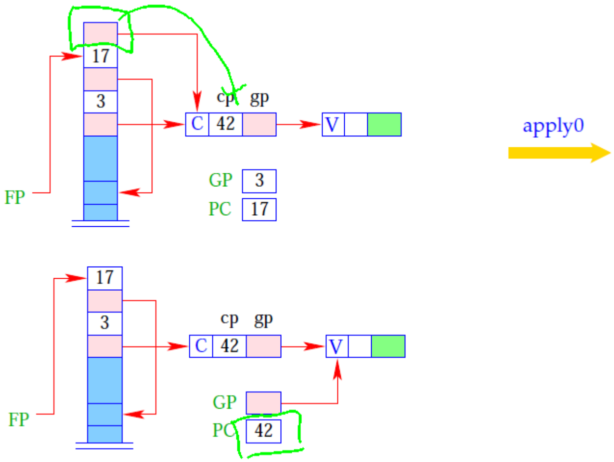
h = S[SP]; SP--;
GP = h→gp; PC = h→cp;

We thus obtain for 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.



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:

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

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

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:

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

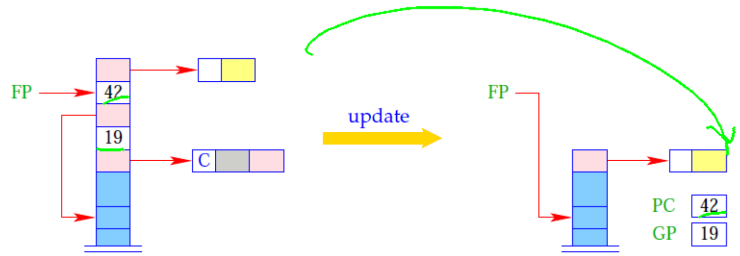
where $\{z_0, \dots, z_{g-1}\} = \mathit{free}(e)$ and $\rho' = \{z_i \mapsto (G, i) \mid i = 0, \dots, g-1\}$.

165

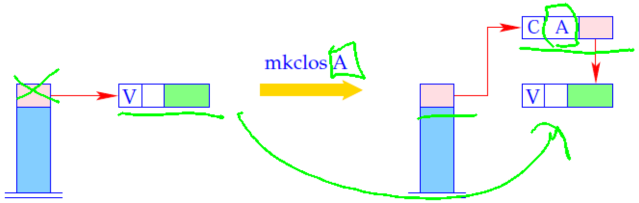
In fact, the instruction update is the combination of the two actions:

popenv
rewrite 1

It overwrites the closure with the computed value.



- 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]);

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:

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

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

Example:

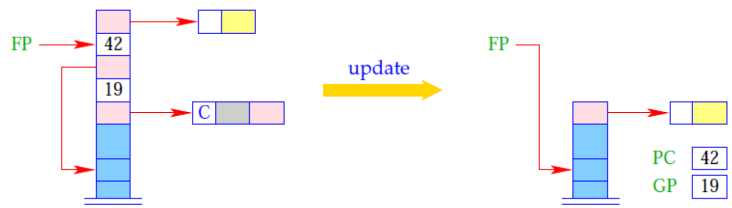
Consider $e \equiv a * a$ with $\rho = \{a \mapsto (L, 0)\}$ and $\text{sd} = 1$. We obtain:

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:	...

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

popenv
rewrite 1

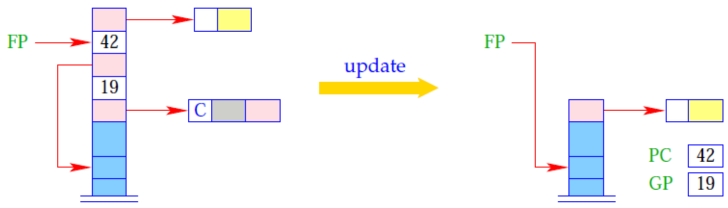
It overwrites the closure with the computed value.



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

popenv
rewrite 1

It overwrites the closure with the computed value.



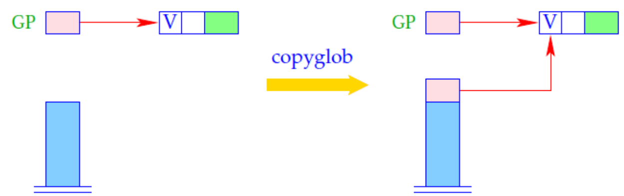
21 Optimizations I: Global Variables

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* ...

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

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



SP++;
S[SP] = GP;

- 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: $e \equiv \textbf{let rec}\; a = b \;\textbf{and}\; b = 7 \;\textbf{in}\; a.$ $\quad \text{code}_V\; e\; \emptyset\; 0$

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

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 $x\ \rho$ sd	mkclos A	A:	pushglob 0		update
mkvec 1	jump B		eval	B:	...

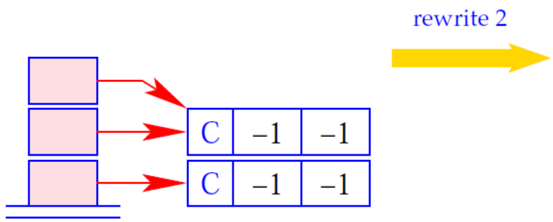
Example: $e \equiv \textbf{let rec}\ a = b\ \textbf{and}\ b = 7\ \textbf{in}\ a.$ $\quad \text{code}_V\ e\ \emptyset\ 0$

produces:

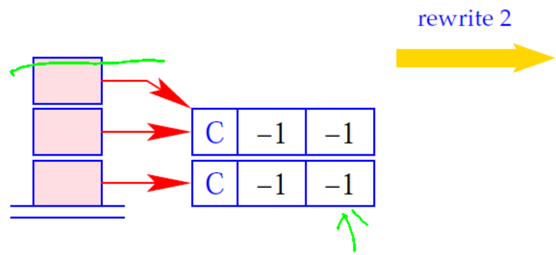
0	alloc 2	3	rewrite 2	3	mkbasic	2	pushloc 1
2	pushloc 0	2	loadc 7	3	rewrite 1	3	eval
						3	slide 2

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

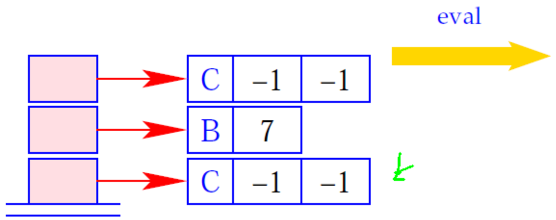
0	alloc 2	3	rewrite 2	3	mkbasic	2	pushloc 1
2	pushloc 0	2	loadc 7	3	rewrite 1	3	eval
						3	slide 2



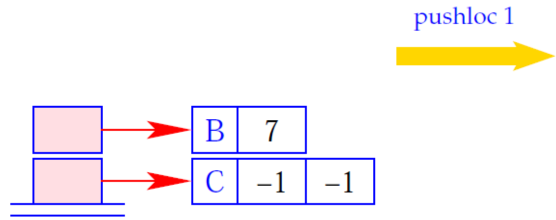
0	alloc 2	3	rewrite 2	3	mkbasic	2	pushloc 1
2	pushloc 0	2	loadc 7	3	rewrite 1	3	eval
						3	slide 2



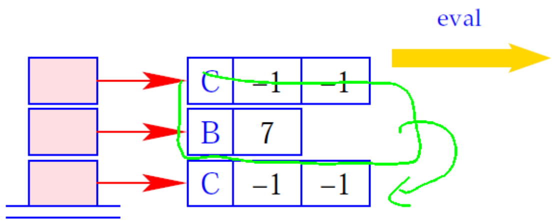
0 alloc 2
2 pushloc 0
3 rewrite 2
2 loadc 7
3 mkbasic
3 rewrite 1
2 pushloc 1
3 eval
3 slide 2



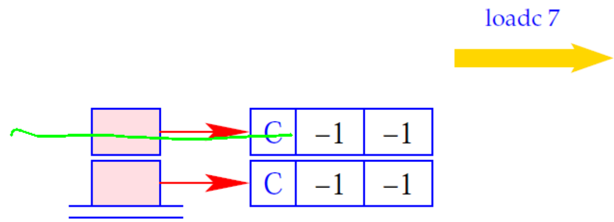
0 alloc 2
2 pushloc 0
3 rewrite 2
2 loadc 7
3 mkbasic
3 rewrite 1
2 pushloc 1
3 eval
3 slide 2



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



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



Apparently, this optimization was not quite correct :-(

The Problem:

Binding of variable y to variable x before x's dummy node is replaced!!

$\Longrightarrow$

The Solution:

cyclic definitions: reject sequences of definitions like
let $a = b; \ldots b = a$ **in** $\ldots$.

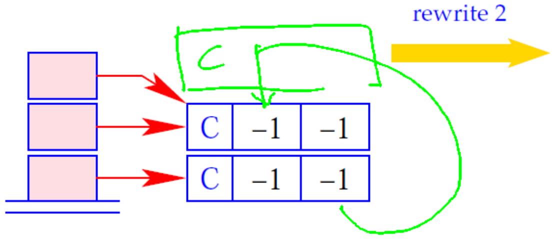
acyclic definitions: order the definitions $y = x$ such that the dummy node for the right side of x is already overwritten.

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

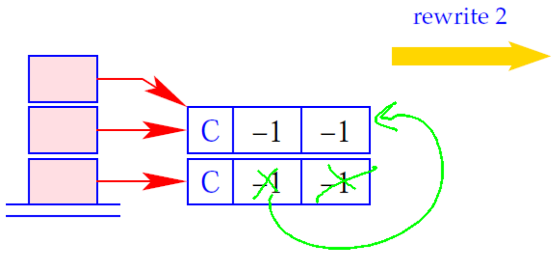
Segmentation Fault !!

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

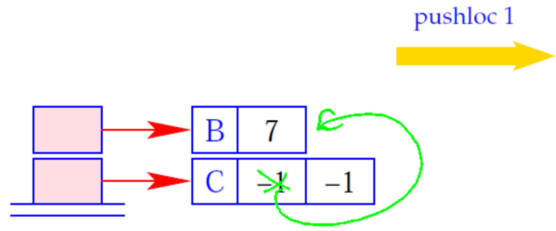
0	alloc 2	3	rewrite 2	3	mkbasic	2	pushloc 1
2	pushloc 0	2	loadc 7	3	rewrite 1	3	eval
						3	slide 2



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:

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

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.

$$\begin{aligned} \text{code}\ e \quad &= \quad \text{code}_V\ e\ \emptyset\ 0 \\ & \qquad \text{halt} \end{aligned}$$

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** $b \rightarrow a + b$ **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