For CBN, we obtain:

\[
\text{code}_v e \rho s d = \begin{cases} 
\text{alloc} \ n & \text{// allocates local variables} \\
\text{code}_v \ e_1 \ \rho' (s d + n) \\
\text{rewrite} \ n \\
\cdots \\
\text{code}_v \ e_n \ \rho' (s d + n) \\
\text{rewrite} \ 1 \\
\text{code}_v \ e_0 \ \rho' (s d + n) \\
\text{slide} \ n & \text{// deallocates local variables}
\end{cases}
\]

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

In the case of CBV, we also use \text{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!

`eval` can be decomposed into small actions:

```plaintext
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 contrast to `mark A`, `mark0` dumps the current `PC`.
- The difference between `apply` and `apply0` is that no argument vector is put on the stack.

20 Closures and their Evaluation

- Closures are needed for the implementation of CBN and for functional parameters.
- 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`.

```plaintext
S[SP+1] = GP;
S[SP+2] = FP;
S[SP+3] = PC;
FP = SP = SP + 3;
```
eval can be decomposed into small actions:

\[
\text{eval} = \begin{cases} 
\text{mark0}; & \text{// allocation of the stack frame} \\
\text{pushloc 3}; & \text{// copying of the reference} \\
\text{apply0}; & \text{// corresponds to apply} 
\end{cases}
\]

- 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 contrast to mark A, mark0 dumps the current PC.
- The difference between apply and apply0 is that no argument vector is put on the stack.

We thus obtain for the instruction eval:

\[
\text{h = S[SP]; SP}; \text{--} \\
\text{GP} = h \rightarrow \text{gp}; \\
\text{PC} = h \rightarrow \text{cp};
\]

\[
\text{h = S[SP]; SP}; \text{--} \\
\text{GP} = h \rightarrow \text{gp}; \\
\text{PC} = h \rightarrow \text{cp};
\]

\[
\text{h = S[SP]; SP}; \text{--} \\
\text{GP} = h \rightarrow \text{gp}; \\
\text{PC} = h \rightarrow \text{cp};
\]
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 $e$:

$$\text{code}_C~e~\rho~sd = \begin{cases} 
\text{getvar } z_0 \rho \text{ sd} \\
\text{getvar } z_1 \rho \text{ (sd + 1)} \\
\vdots \\
\text{getvar } z_{g-1} \rho \text{ (sd + g - 1)} \\
\text{mkvec } g \\
\text{mkclos } A \\
\text{jump } B \\
A : \text{code}_C~e~\rho'~\emptyset \\
\text{update } \\
B : \ldots
\end{cases}$$

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

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

1. pushloc 1
2. mkvec 1
3. mkclose A
4. jump B

A: pushglob 0
eval
getbasic
jump B
B: ...

glob 0
mul
mkbasic
update

eval

Example:

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

1. pushloc 1
2. mkvec 1
3. mkclose A
4. jump B

A: pushglob 0
eval
glob 0
mul
mkbasic
update

B: ...

glob 0
mkbasic
update

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 mkclose 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.

```
FP 42  
       |  
       v  
19 C   
     |  
     v  
PC GP 42 19
```

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 \ \mathsf{sd} = \begin{array}{l}
\text{getvar} \ z_0 \ \rho \ \mathsf{sd} \\
\text{getvar} \ z_1 \ \rho \ (\mathsf{sd} + 1) \\
\vdots \\
\text{getvar} \ z_{g-1} \ \rho \ (\mathsf{sd} + g - 1) \\
\text{mkvec} \ \mathsf{g} \\
\text{mkclsc} \ A \\
\text{jump} \ B \\
A : \ \text{code}_c \ e' \ 0 \\
\text{update} \\
B : \ \ldots
\end{array}
```

where $\{z_0, \ldots, z_{g-1}\} = \text{free}(e)$ 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 \(\implies\) 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:

\[
\text{code}_{\mathcal{C}} \, b \, \rho \, s \, d = \text{code}_{\mathcal{V}} \, b \, \rho \, s \, d = \text{loadc} \, b \\
\text{mkbasic}
\]

This replaces:

\[
\begin{align*}
\text{mkvec} \, 0 & \quad \text{jump} \, \beta \\
\text{mkvec} \, A & \quad \text{loadc} \, b \\
\text{mklos} \, A & \quad \text{update} \\
\end{align*}
\]

---

**Variables:**

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

\[
\text{code}_{\mathcal{C}} \, x \, \rho \, s \, d = \text{getvar} \, x \, \rho \, s \, d
\]

This replaces:

\[
\begin{align*}
\text{getvar} \, x \, \rho \, s \, d & \quad \text{mklos} \, A \\
\text{mkvec} \, 1 & \quad \text{jump} \, \beta \\
\text{mklos} \, A & \quad \text{pushglob} \, 0 \\
\text{update} & \quad \text{eval} \\
\end{align*}
\]
Example:

Consider \( e = \text{let rec } a = b \text{ and } b = 7 \text{ in } a. \)

code: \( e 0 0 \) produces:

\[
\begin{align*}
0 & \quad \text{alloc 2} & 3 & \quad \text{rewrite 2} & 3 & \quad \text{mkbasic} & 2 & \quad \text{pushloc 1} \\
2 & \quad \text{pushloc 0} & 2 & \quad \text{loadc 7} & 3 & \quad \text{rewrite 1} & 3 & \quad \text{eval} \\
& & & & & 3 & \quad \text{slide 2}
\end{align*}
\]

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

Segmentation Fault!!
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:
\[
\text{code}_c \ (\text{fun } x_0 \ldots x_{k-1} \to e) \ \rho \ \text{sd} = \ \text{code}_v \ (\text{fun } x_0 \ldots x_{k-1} \to e) \ \rho \ \text{sd}
\]

23 The Translation of a Program Expression

Execution of a program \( e \) starts with
\[
\begin{align*}
\text{PC} &= 0 & \text{SP} &= \text{FP} = \text{GP} = -1
\end{align*}
\]

The expression \( e \) must not contain free variables.
The value of \( e \) should be determined and then a \( \text{halt} \) instruction should be executed.

\[
\begin{align*}
\text{code } e &= \ \text{code}_v \ e \ \parallel \ 0 \\
&\quad \text{halt}
\end{align*}
\]
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.

Solution:

Disentangle the Spaghetti code in a subsequent optimization phase