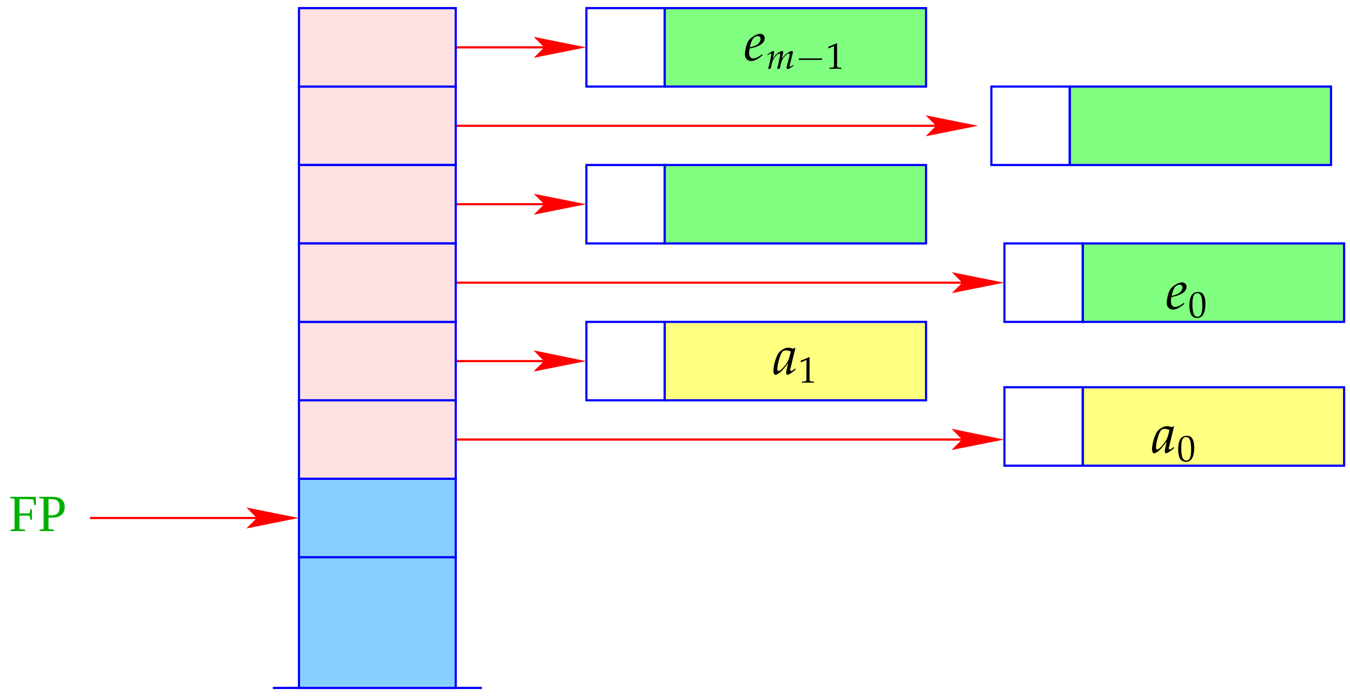
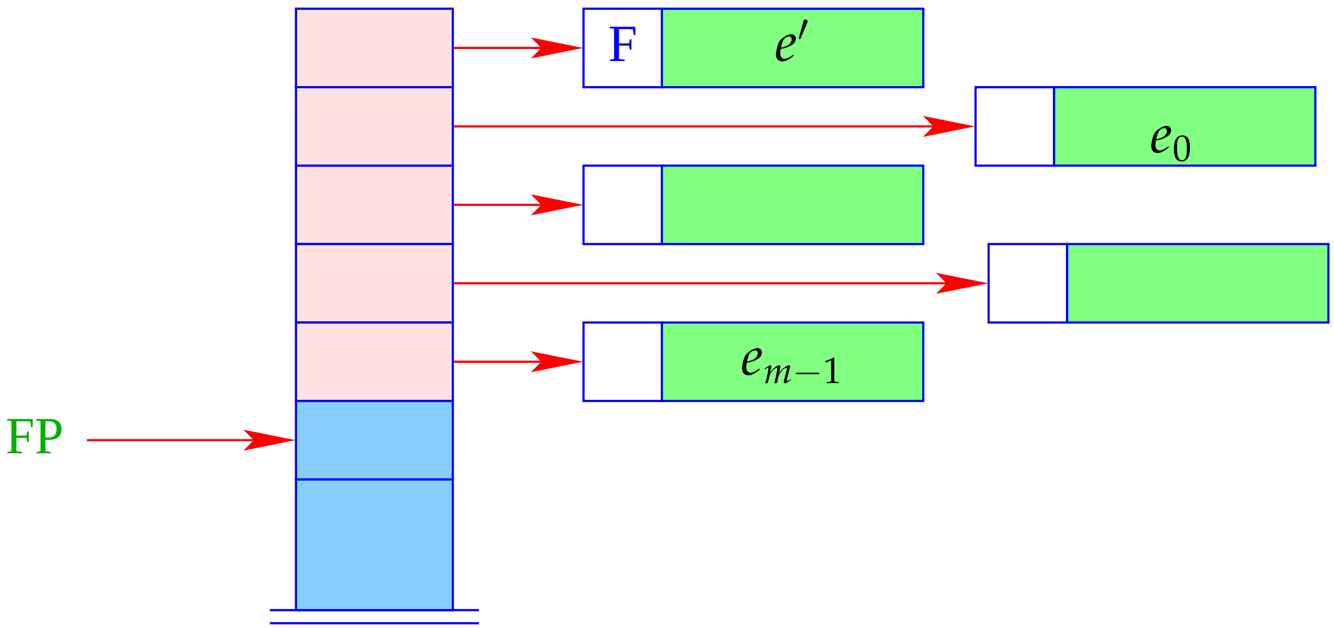
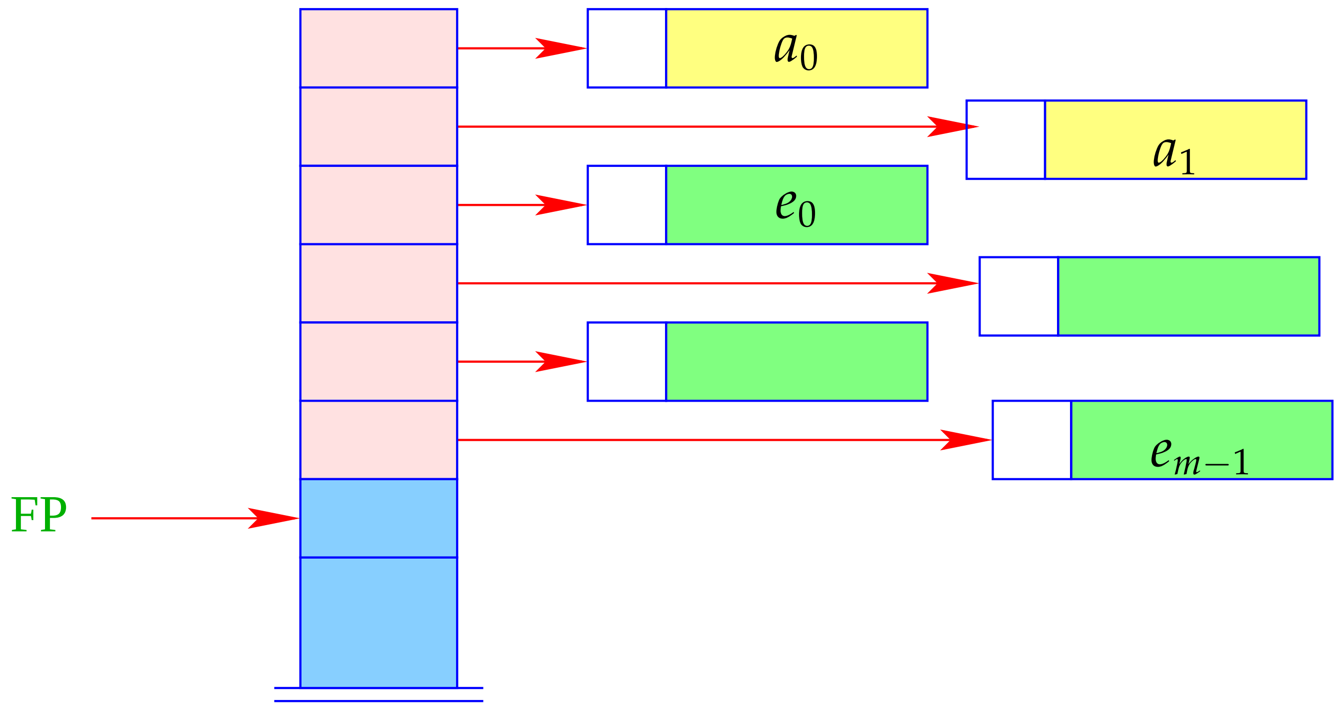
– If $e'$ evaluates to a function, which has already been partially applied to the parameters $a_0, \ldots, a_{k-1}$, these have to be sneaked in underneath $e_0$:



## Alternative:



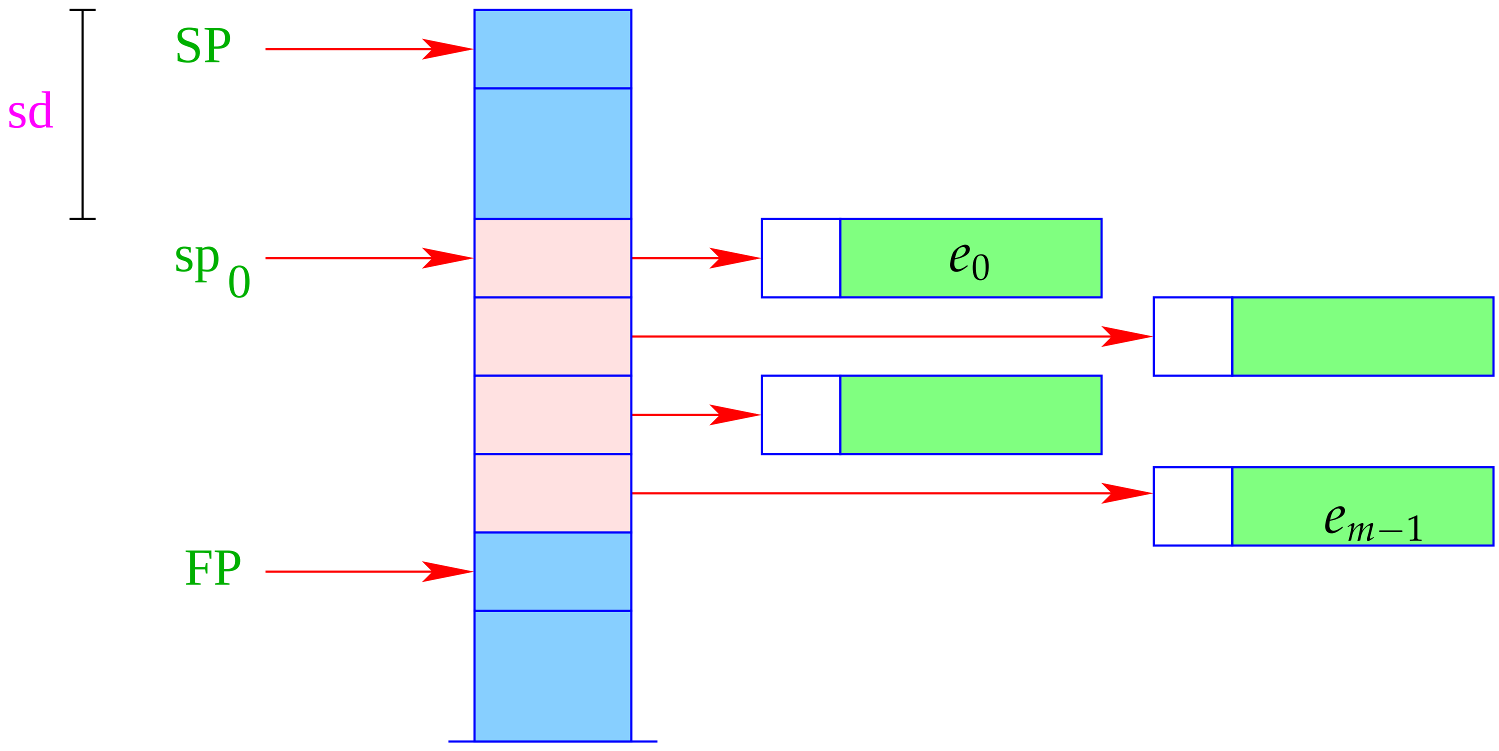

+ The further arguments $a_0, \ldots, a_{k-1}$ and the local variables can be allocated above the arguments.




– Addressing of arguments and local variables relative to FP is no more possible. (Remember: $m$ is unknown when the function definition is translated.)

## Way out:

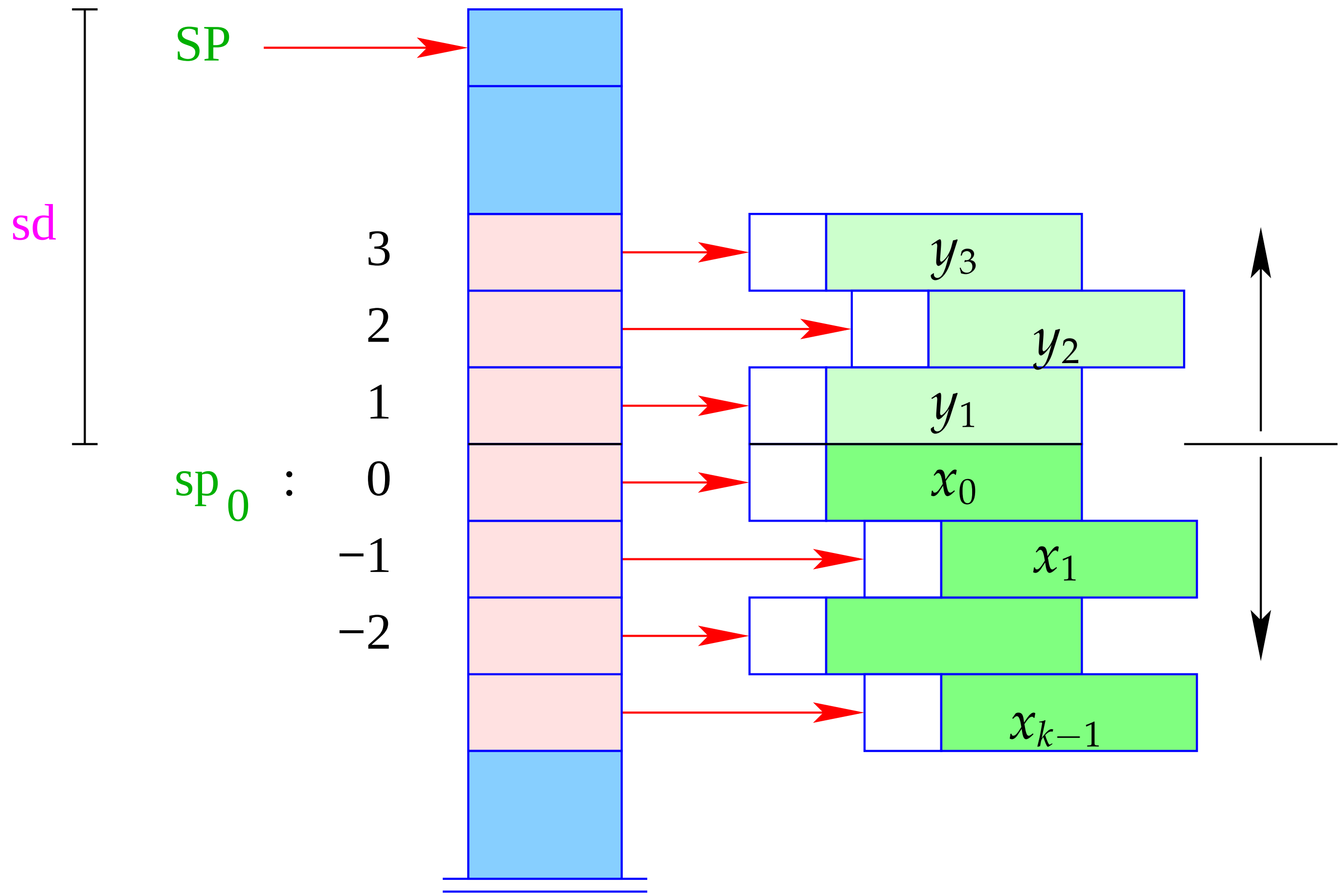
- We address both, arguments and local variables, relative to the stack pointer SP !!!
- However, the stack pointer changes during program execution...



- The differerence between the current value of SP and its value $\mathrm{sp}_0$ at the entry of the function body is called the stack distance, sd.
- Fortunately, this stack distance can be determined at compile time for each program point, by simulating the movement of the SP.
- The formal parameters $x_0, x_1, x_2, \ldots$ successively receive the non-positive relative addresses $0, -1, -2, \ldots$, i.e., $\rho\, x_i = (L, -i)$.
- The absolute address of the $i$-th formal parameter consequently is

$$\mathrm{sp}_0 - i = (\mathrm{SP} - \mathrm{sd}) - i$$

- The local **let**-variables $y_1, y_2, y_3, \ldots$ will be successively pushed onto the stack:




- The $y_i$ have positive relative addresses $1, 2, 3, \ldots$, that is: $\rho\ y_i = (L, i)$.
- The absolute address of $y_i$ is then $sp_0 + i = (SP - sd) + i$

With CBN, we generate for the access to a variable:

$$\text{code}_V\ x\ \rho\ \text{sd} \quad = \quad \begin{array}{l} \text{getvar}\ x\ \rho\ \text{sd} \\ \text{eval} \end{array}$$

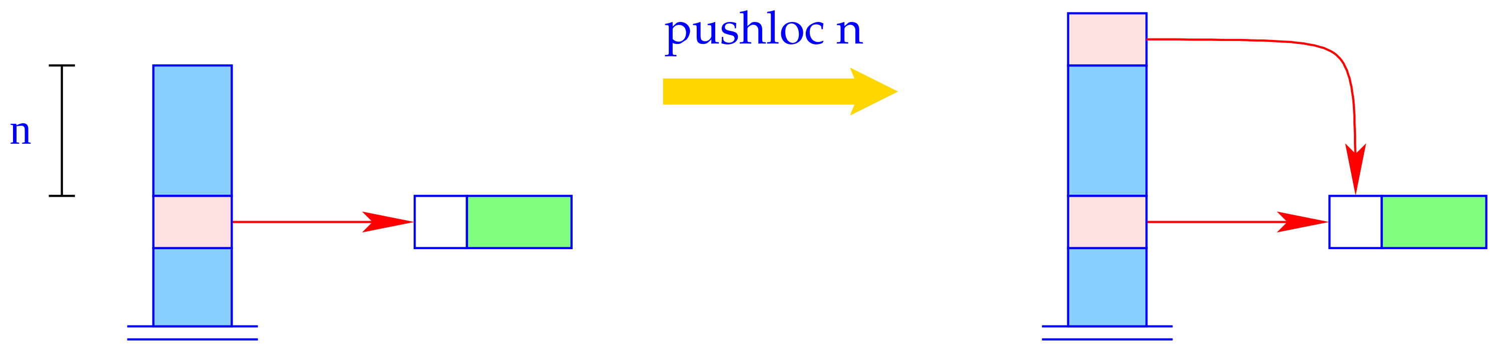
The instruction eval checks, whether the value has already been computed or whether its evaluation has to yet to be done ($\Longrightarrow$ will be treated later :-)

With CBV, we can just delete eval from the above code schema.

The (compile-time) macro getvar is defined by:

$$\begin{array}{lll} \text{getvar}\ x\ \rho\ \text{sd} & = & \textbf{let}\ (t, i) = \rho\ x\ \textbf{in} \\ & & \textbf{case}\ t\ \textbf{of} \\ & & \quad L \Rightarrow \text{pushloc}\ (\text{sd} - i) \\ & & \quad G \Rightarrow \text{pushglob i} \\ & & \textbf{end} \end{array}$$

The access to local variables:




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

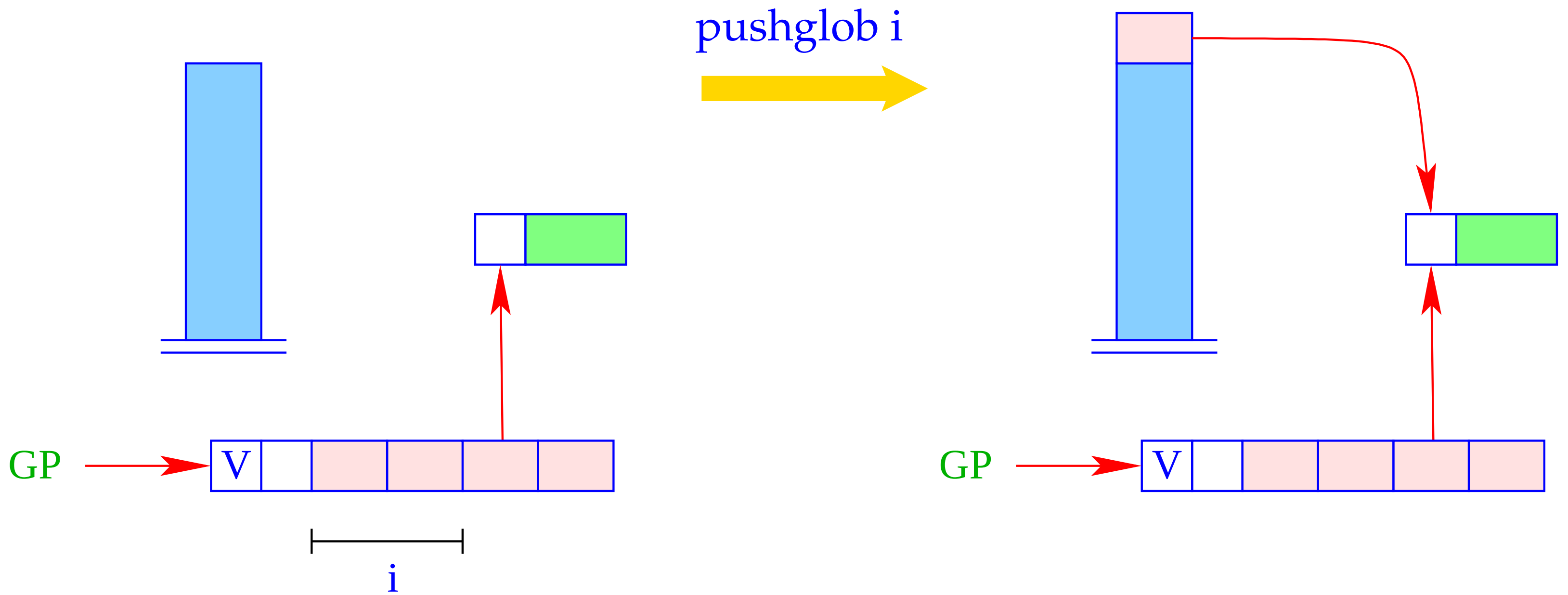
## Correctness argument:

Let sp and sd be the values of the stack pointer resp. stack distance before the execution of the instruction. The value of the local variable with address $i$ is loaded from $S[a]$ with

$$a = \mathrm{sp} - (\mathrm{sd} - i) = (\mathrm{sp} - \mathrm{sd}) + i = \mathrm{sp}_0 + i$$

... exactly as it should be :-)

The access to global variables is much simpler:




SP = SP + 1;
S[SP] = GP→v[i];

## Example:

Regard $e \equiv (b + c)$ for $\rho = \{b \mapsto (L, 1), c \mapsto (G, 0)\}$ and $\mathit{sd} = 1$.

With CBN, we obtain:

| | | | | | |
|---|---|---|---|---|---|
| $\text{code}_V\ e\ \rho\ 1$ | $=$ | $\text{getvar}\ b\ \rho\ 1$ | $=$ | 1 | pushloc 0 |
| | | eval | | 2 | eval |
| | | getbasic | | 2 | getbasic |
| | | $\text{getvar}\ c\ \rho\ 2$ | | 2 | pushglob 0 |
| | | eval | | 3 | eval |
| | | getbasic | | 3 | getbasic |
| | | add | | 3 | add |
| | | mkbasic | | 2 | mkbasic |

# 15 let-Expressions

As a warm-up let us first consider the treatment of local variables :-)

Let $e \equiv \textbf{let}\ y_1 = e_1; \ldots; y_n = e_n\ \textbf{in}\ e_0$ be a **let**-expression.

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.

Here, we consider the non-recursive case only, i.e. where $y_j$ only depends on $y_1, \ldots, y_{j-1}$. We obtain for CBN:

$$
\begin{array}{lcl}
\text{code}_V\ e\ \rho\ \text{sd} & = & \text{code}_C\ e_1\ \rho\ \text{sd} \\
& & \text{code}_C\ e_2\ \rho_1\ (\text{sd}+1) \\
& & \ldots \\
& & \text{code}_C\ e_n\ \rho_{n-1}\ (\text{sd}+n-1) \\
& & \text{code}_V\ e_0\ \rho_n\ (\text{sd}+n) \\
& & \text{slide n} \qquad\qquad \text{// deallocates local variables}
\end{array}
$$

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

In the case of CBV, we use $\text{code}_V$ for the expressions $e_1, \ldots, e_n$.

## Warning!

All the $e_i$ must be associated with the same binding for the global variables!

## Example:

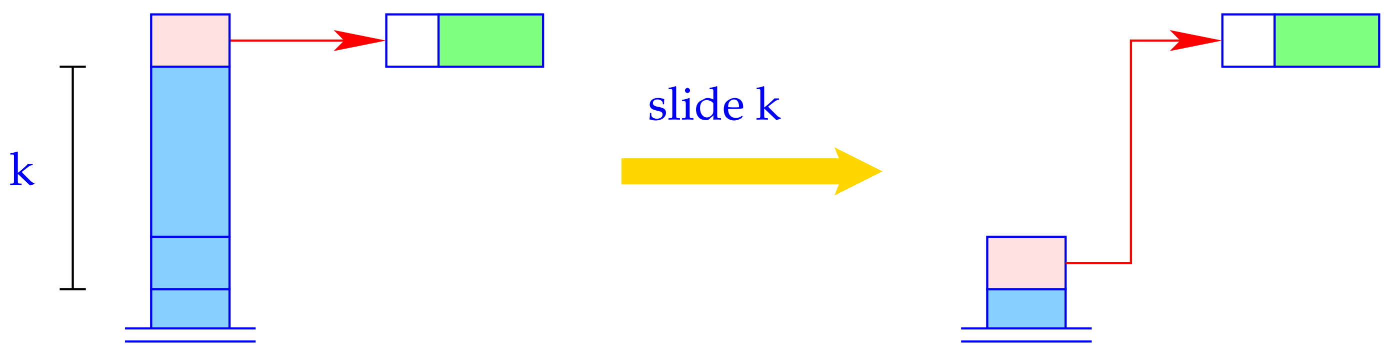
Consider the expression

$$e \equiv \textbf{let}\ a = 19; b = a * a\ \textbf{in}\ a + b$$

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

| | | | | | |
|---|---|---|---|---|---|
| 0 | loadc 19 | 3 | getbasic | 3 | pushloc 1 |
| 1 | mkbasic | 3 | mul | 4 | getbasic |
| 1 | pushloc 0 | 2 | mkbasic | 4 | add |
| 2 | getbasic | 2 | pushloc 1 | 3 | mkbasic |
| 2 | pushloc 1 | 3 | getbasic | 3 | slide 2 |

The instruction slide k deallocates again the space for the locals:




S[SP-k] = S[SP];
SP = SP - k;

# 16 Function Definitions

The definition of a function $f$ requires code that allocates a functional value for $f$ in the heap. This happens in the following steps:

- Creation of a Global Vector with the binding of the free variables;
- Creation of an (initially empty) argument vector;
- Creation of an F-Object, containing references to theses vectors and the start address of the code for the body;

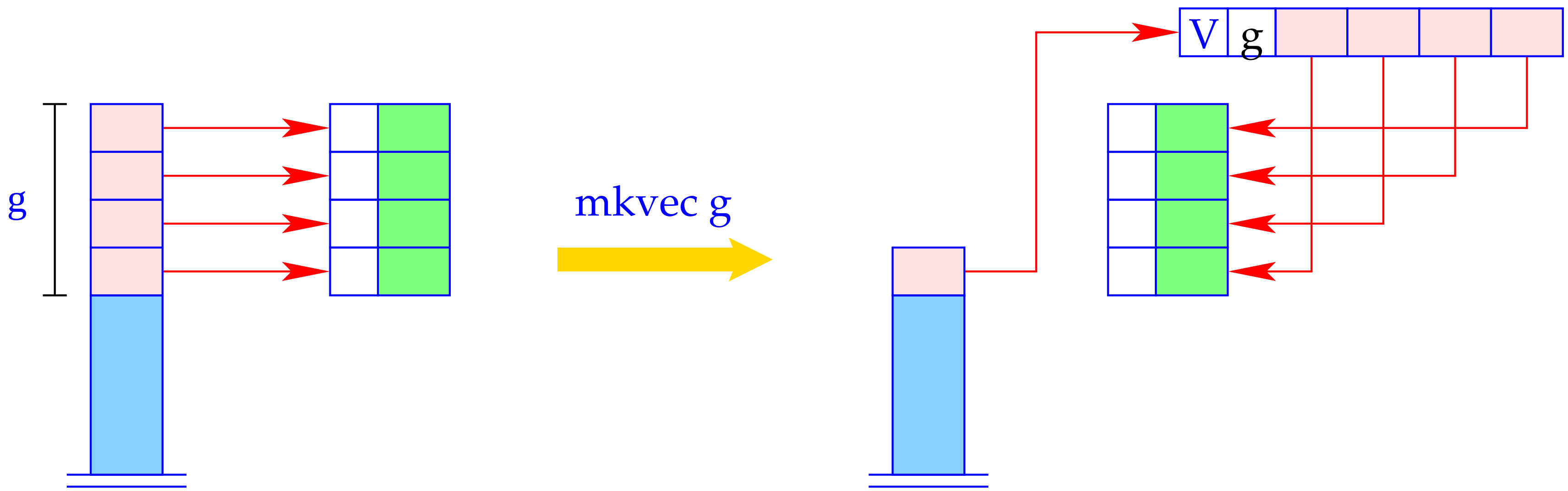
Separately, code for the body has to be generated.

Thus:

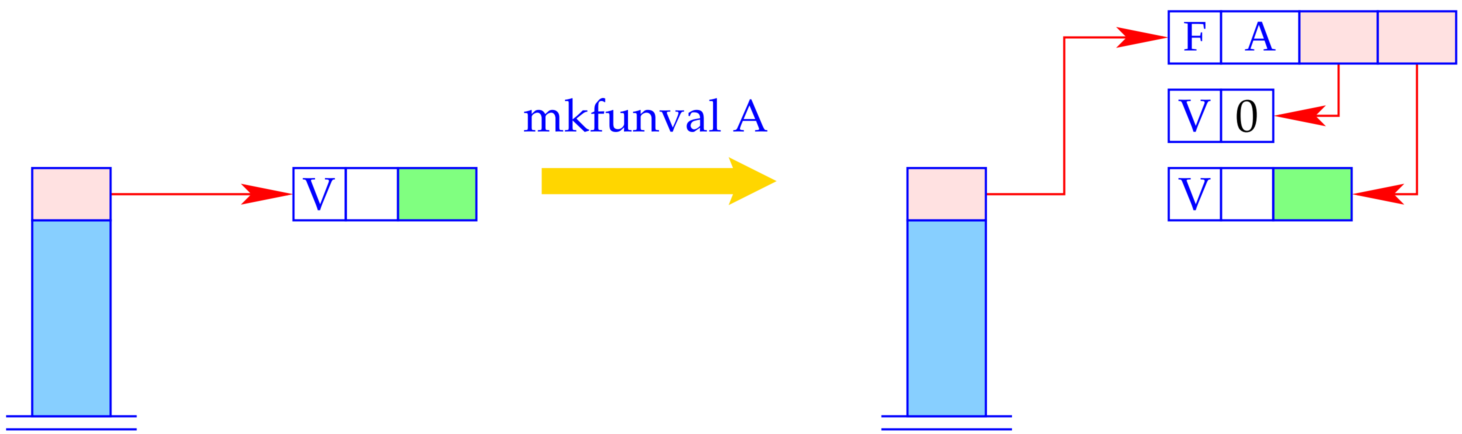
$$
\begin{array}{rcll}
\mathrm{code}_V\ (\mathbf{fn}\ x_0, \ldots, x_{k-1} \Rightarrow e)\ \rho\ \mathrm{sd} & = & & \mathrm{getvar}\ z_0\ \rho\ \mathrm{sd} \\
& & & \mathrm{getvar}\ z_1\ \rho\ (\mathrm{sd}+1) \\
& & & \ldots \\
& & & \mathrm{getvar}\ z_{g-1}\ \rho\ (\mathrm{sd}+g-1) \\
& & & \mathrm{mkvec\ g} \\
& & & \mathrm{mkfunval\ A} \\
& & & \mathrm{jump\ B} \\
& & \mathrm{A:} & \mathrm{targ\ k} \\
& & & \mathrm{code}_V\ e\ \rho'\ 0 \\
& & & \mathrm{return\ k} \\
& & \mathrm{B:} & \ldots
\end{array}
$$

where $\{z_0, \ldots, z_{g-1}\} = \mathit{free}(\mathbf{fn}\ x_0, \ldots, x_{k-1} \Rightarrow e)$

and $\rho' = \{x_i \mapsto (L, -i) \mid i = 0, \ldots, k-1\} \cup \{z_j \mapsto (G, j) \mid j = 0, \ldots, g-1\}$




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




a = new (V,0);
S[SP] = new (F, A, a, S[SP]);

## Example:

Regard $f \equiv \textbf{fn}\ b \Rightarrow a + b$ for $\rho = \{a \mapsto (L, 1)\}$ and sd $= 1$.

$\text{code}_V\ f\ \rho\ 1$ produces:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | | pushloc 0 | 0 | pushglob 0 | 2 | | getbasic |
| 2 | | mkvec 1 | 1 | eval | 2 | | add |
| 2 | | mkfunval A | 1 | getbasic | 1 | | mkbasic |
| 2 | | jump B | 1 | pushloc 1 | 1 | | return 1 |
| 0 | A : | targ 1 | 2 | eval | 2 | B : | ... |

The secrets around targ k and return k will be revealed later :-)

# 17 Function Application

Function applications correspond to function calls in C.
The necessary actions for the evaluation of $e' \, e_0 \, \ldots \, e_{m-1}$ are:

- Allocation of a stack frame;
- Transfer of the actual parameters , i.e. with:
  CBV: Evaluation of the actual parameters;
  CBN: Allocation of closures for the actual parameters;
- Evaluation of the expression $e'$ to an F-object;
- Application of the function.

Thus for CBN:

$$
\begin{array}{rcll}
\text{code}_V\ (e'\ e_0\ \ldots\ e_{m-1})\ \rho\ \text{sd} & = & \text{mark A} & \text{// Allocation of the frame} \\
& & \text{code}_C\ e_{m-1}\ \rho\ (\text{sd}+3) & \\
& & \text{code}_C\ e_{m-2}\ \rho\ (\text{sd}+4) & \\
& & \ldots & \\
& & \text{code}_C\ e_0\ \rho\ (\text{sd}+m+2) & \\
& & \text{code}_V\ e'\ \rho\ (\text{sd}+m+3) & \text{// Evaluation of } e' \\
& & \text{apply} & \text{// corresponds to call} \\
& A: & \ldots &
\end{array}
$$

To implement CBV, we use $\text{code}_V$ instead of $\text{code}_C$ for the arguments $e_i$.

Example: For $(f\ 42)$ , $\rho = \{f \mapsto (L, 2)\}$ and $\text{sd} = 2$, we obtain with CBV:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | mark A | 6 | mkbasic | 7 | | apply |
| 5 | loadc 42 | 6 | pushloc 4 | 3 | A : | ... |

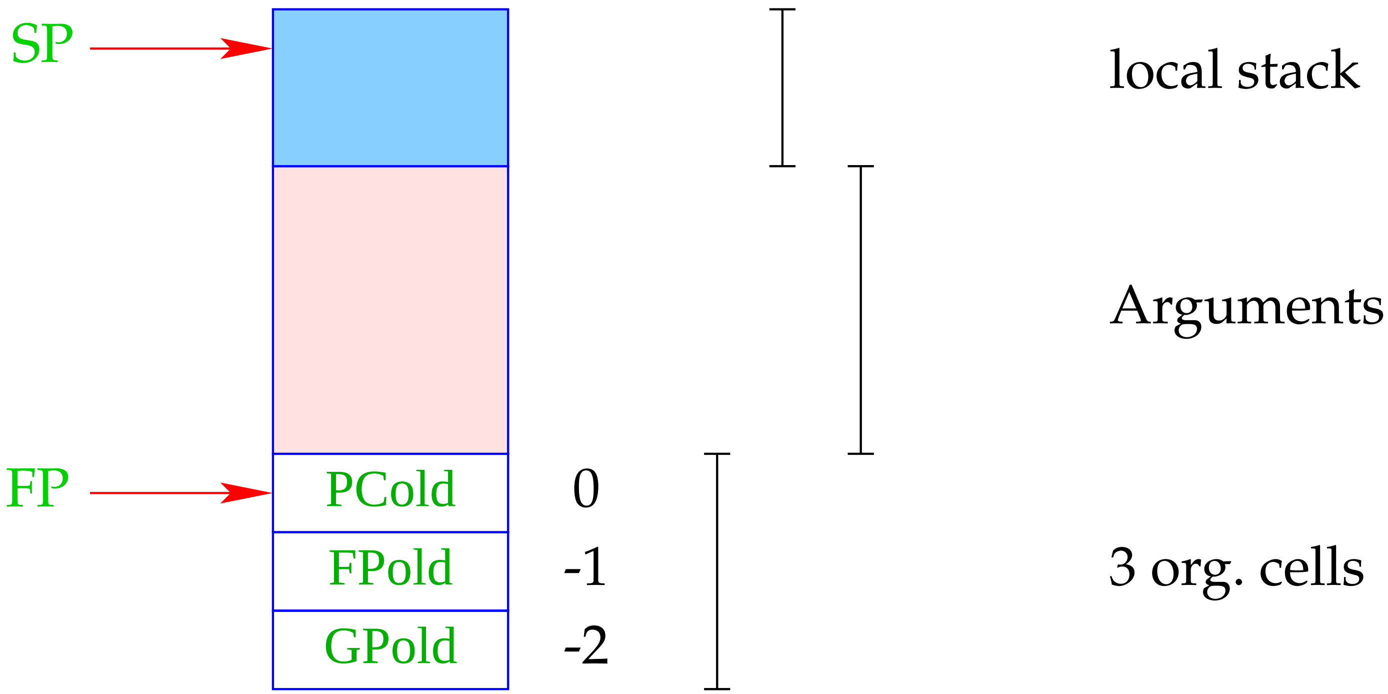
## A Slightly Larger Example:

$$\textbf{let } a = 17;\ f = \textbf{fn } b \Rightarrow a + b \textbf{ in } f\ 42$$

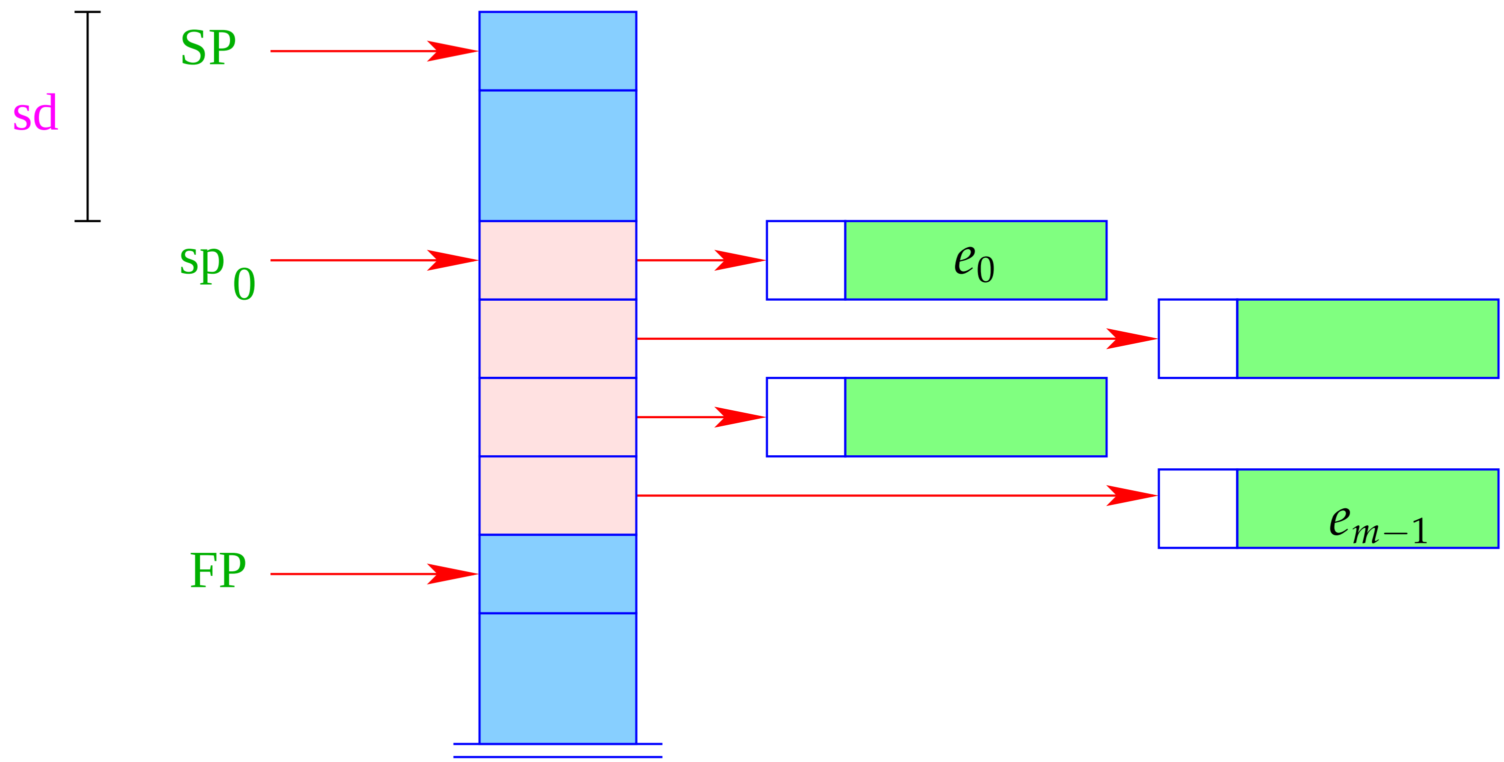
For CBV and $kp = 0$ we obtain:

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

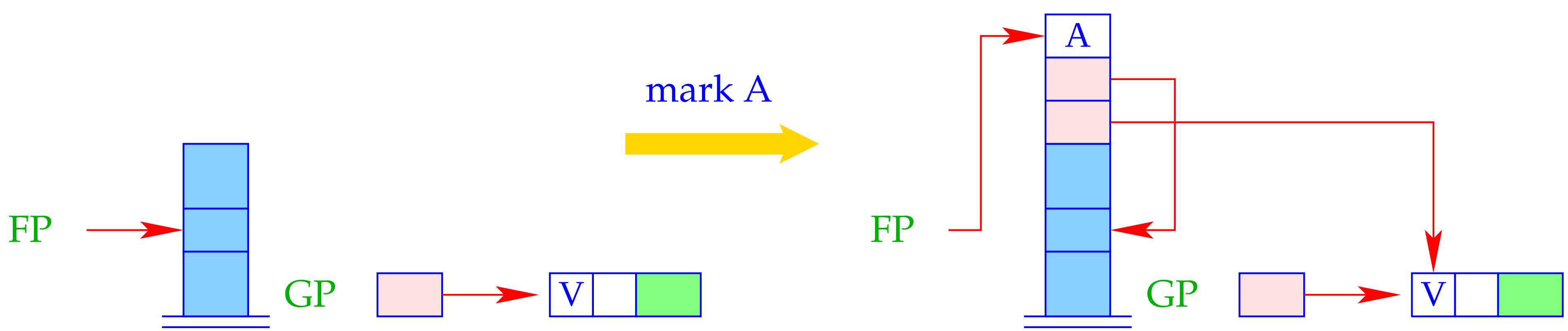
For the implementation of the new instruction, we must fix the organization of a stack frame:



Remember: Addressing of arguments and local variables

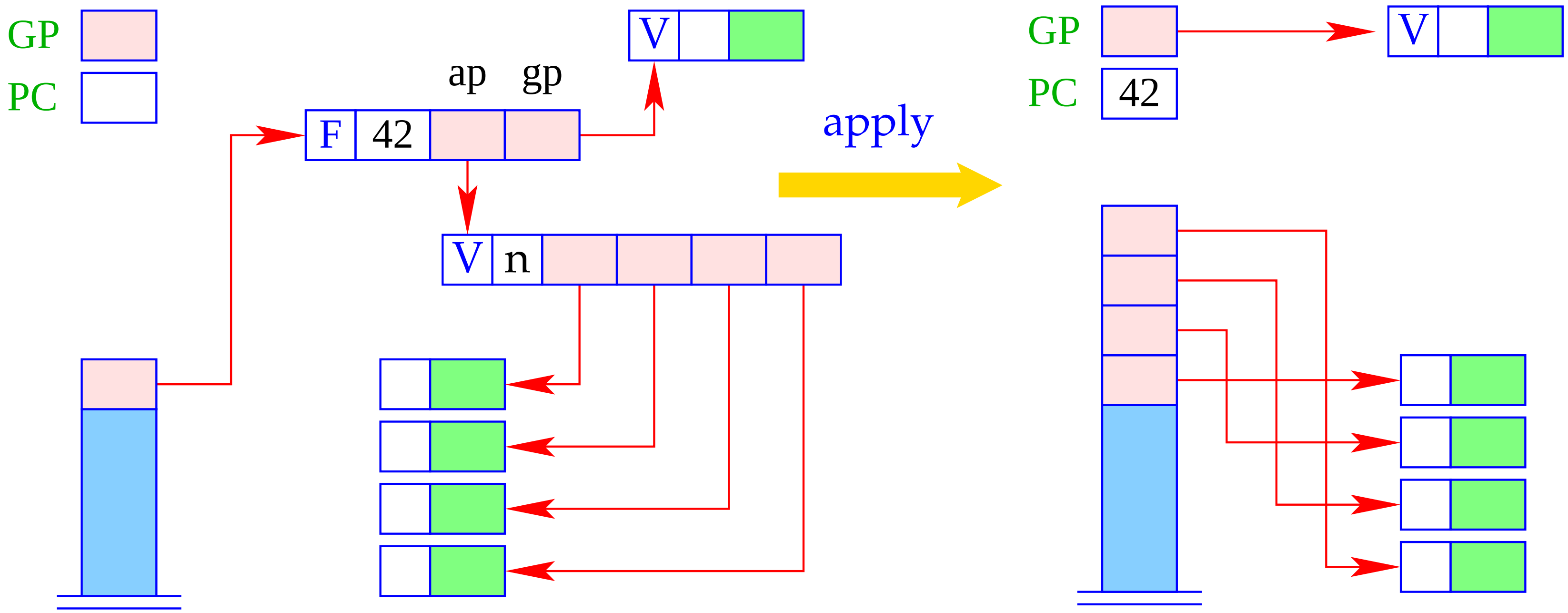


Different from the CMa, the instruction mark A already saves the return address:




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

The instruction apply unpacks the F-object, a reference to which (hopefully) resides on top of the stack, and continues execution at the address given there:




```
h = S[SP];
if (H[h] != (F,_,_))
    Error "no fun";
else {
    GP = h→gp; PC = h→cp;
    for (i=0; i< h→ap→n; i++)
        S[SP+i] = h→ap→v[i];
    SP = SP + h→ap→n – 1;
}
```