MaMa — funktsionaalsete keelte lihtsustatud abstraktne masin

Funktsionaalne keel PuF

Käsitleme funktsionaalset mini-keelt PuF ("Pure Functions").

Programm on avaldis e kujul:

$$
\begin{array}{rcl}
e & ::= & b \mid x \mid (\Box_1\, e) \mid (e_1\, \Box_2\, e_2) \\
& \mid & (\textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_3) \\
& \mid & (e'\ e_0\ \ldots\ e_{k-1}) \\
& \mid & (\textbf{fn } x_0, \ldots, x_{k-1} \Rightarrow e) \\
& \mid & (\textbf{let } x_1 = e_1; \ldots; x_n = e_n \textbf{ in } e_0) \\
& \mid & (\textbf{letrec } x_1 = e_1; \ldots; x_n = e_n \textbf{ in } e_0)
\end{array}
$$

- Lihtsuse mõttes on ainsaks baastüübiks `int`.
- Hiljem lisame ka andmestruktuurid.

Funktsionaalne keel PuF

Näide: faktoriaali arvutava funktsiooni saab defineerida alljärgnevalt:

$$\begin{aligned} \text{fac} \quad = \quad & \textbf{fn}\ x \Rightarrow \textbf{if}\ x \leq 1\ \textbf{then}\ 1 \\ & \textbf{else}\ x \cdot \text{fac}\,(x-1) \end{aligned}$$

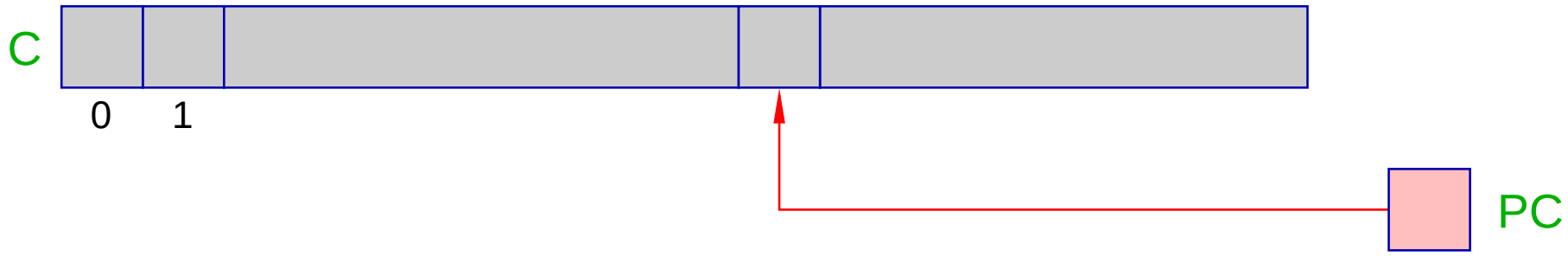
Funktsionaalsetes keeltes on kasutusel kaks erinevat semantikat:

CBV: argumendid väärtustatakse enne nende funktsioonile edastamist (näit. SML);

CBN: argumendid edastatakse funktsioonile ilma väärtustamata ning väärtustatakse alles siis kui nende väärtust on vaja (näit. Haskell).

MaMa arhitektuur

Kood:

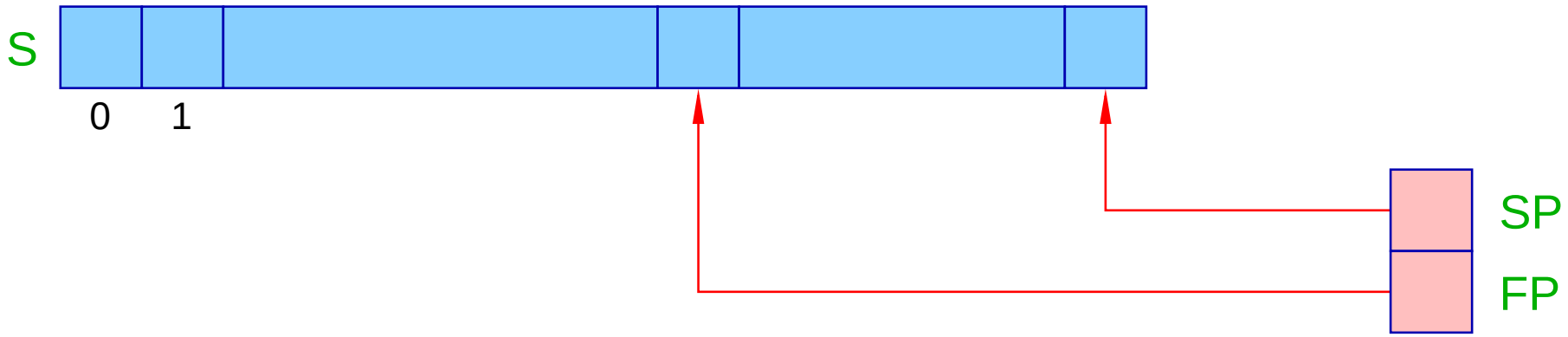


C = Code-store — mälupiirkond MaMa programmikoodi hoidmiseks; igas pesas on täpselt üks abstraktse masina käsk.

PC = Program Counter — viitab järgmisena täidetavale käsule.

MaMa arhitektuur

Magasin:



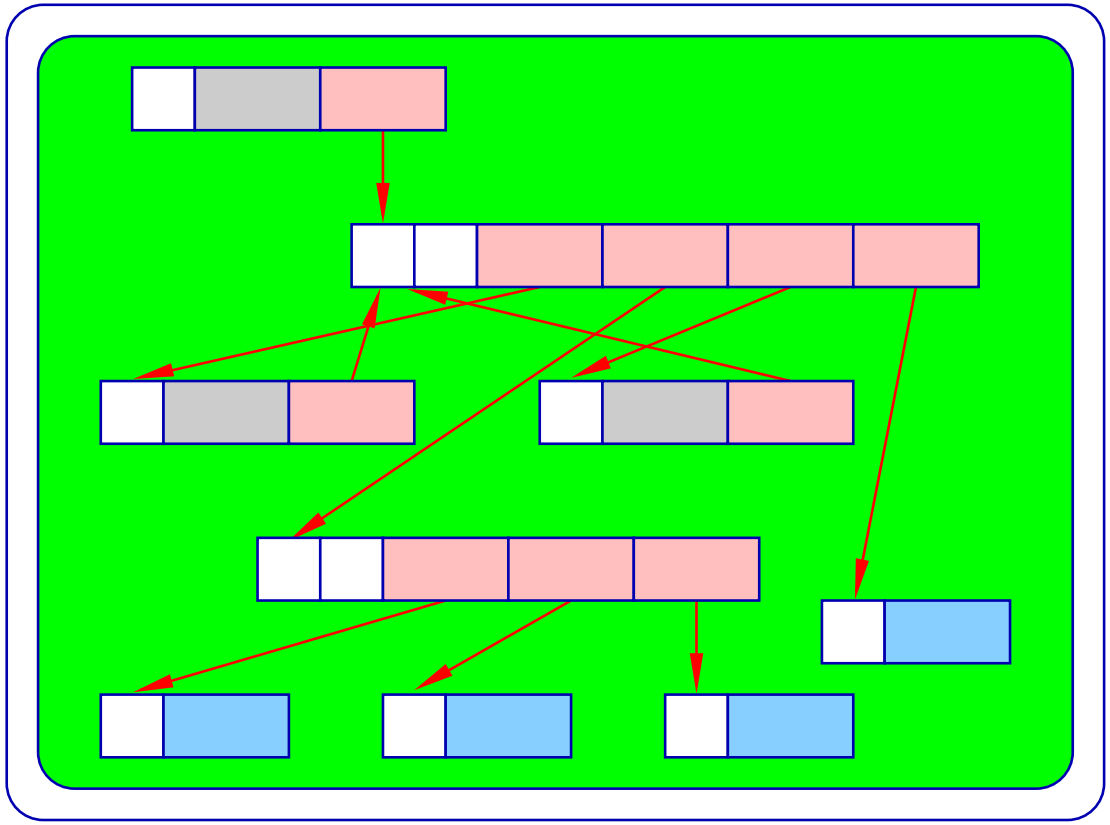
S = Stack — igas pesas võib olla kas baasväärtus või aadress;

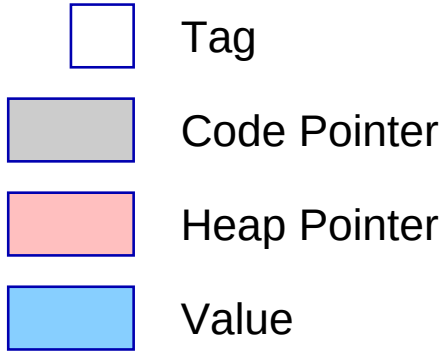
SP = Stack-Pointer — viitab tipmisele täidetud pesale;

FP = Frame-Pointer — viitab kehtivale freimile.

MaMa arhitektuur

Kuhi:

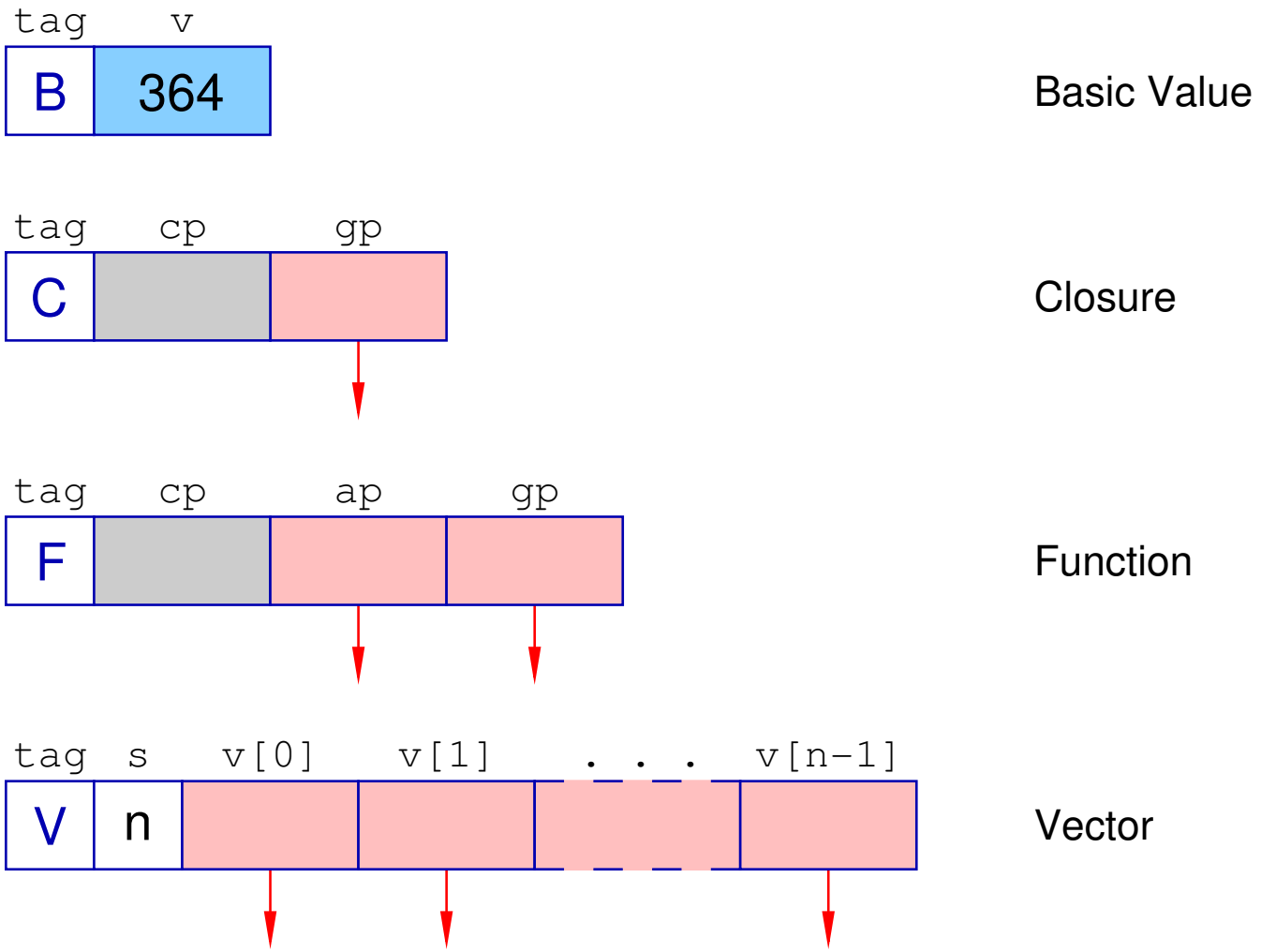




H = Heap — mälupiirkond dünaamiliste andmete hoidmiseks.

MaMa arhitektuur

Kuhjas võivad olla järgmised objektid:



MaMa arhitektuur

Käsk **new**$(tag, args)$ loob etteantud liiki uue kuhjaobjekti ning väljastab viida temale.

Avaldisele e vastava koodi genereerimiseks kasutame kolme erinevat funktsiooni:

- $\text{code}_B\ e$ — leiab avaldise e väärtuse ja salvestab selle magasini tippu (ainult baastüüpide jaoks);
- $\text{code}_V\ e$ — leiab avaldise e väärtuse, salvestab selle kuhja ning väljastab magasini tippu viida vastavale kuhjaobjektile;
- $\text{code}_C\ e$ — ei väärtusta avaldist, vaid salvestab e sulundi (closure) kuhja ning väljastab viida sellele sulundile magasini tippu.

Lihtsate avaldiste transleerimine

Avaldised, mis koosnevad ainult konstantidest, operaatorite rekendustest ja tingimusavaldistest, transleeritakse analoogselt imperatiivsete keeltega:

$$
\begin{array}{lcl}
\text{code}_B\ b\ \rho\ \text{sd} & = & \texttt{loadc b} \\
\text{code}_B\ (\square_1\, e)\ \rho\ \text{sd} & = & \text{code}_B\ e\ \rho\ \text{sd} \\
 & & \text{op}_1 \\
\text{code}_B\ (e_1\, \square_2\, e_2)\ \rho\ \text{sd} & = & \text{code}_B\ e_1\ \rho\ \text{sd} \\
 & & \text{code}_B\ e_2\ \rho\ (\text{sd}+1) \\
 & & \text{op}_2
\end{array}
$$

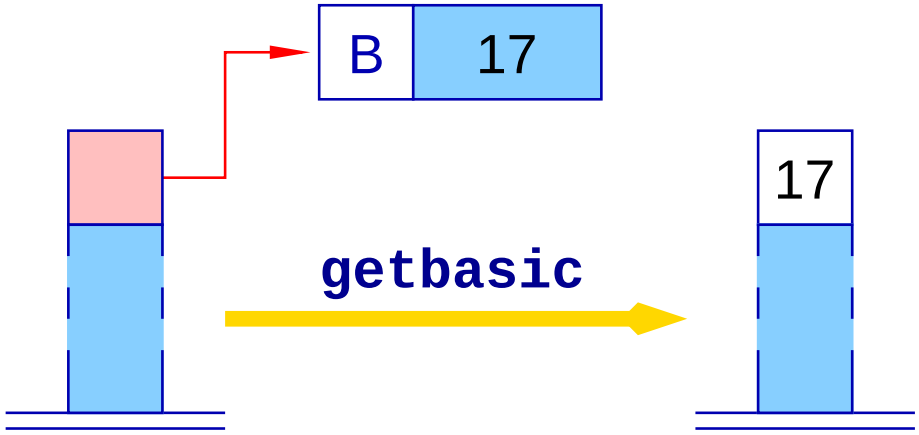
Lihtsate avaldiste transleerimine

$$
\begin{array}{rll}
\text{code}_B\ (\textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2)\ \rho\ \text{sd} & = & \text{code}_B\ e_0\ \rho\ \text{sd} \\
 & & \texttt{jumpz A} \\
 & & \text{code}_B\ e_1\ \rho\ \text{sd} \\
 & & \texttt{jump B} \\
 & \text{A:} & \text{code}_B\ e_2\ \rho\ \text{sd} \\
 & \text{B:} & \ldots
\end{array}
$$

Teiste avaldiste korral arvutame kõigepealt tema väärtuse kuhjas ja seejärel võtame väärtuse väljastatud viida kaudu:

$$
\begin{array}{rcl}
\text{code}_B\ e\ \rho\ \text{sd} & = & \text{code}_V\ e\ \rho\ \text{sd} \\
 & & \texttt{getbasic}
\end{array}
$$

Lihtsate avaldiste transleerimine




```
if (S[SP]→tag ≠ B)
  Error("Not Basic");
else
  S[SP] = S[SP]→v;
```

- ρ tähistab aadresskeskkonda milles avaldisi transleeritakse. Aadresskeskkonnad on kujul:

$$\rho : Vars \to \{L, G\} \times \mathbb{Z}$$

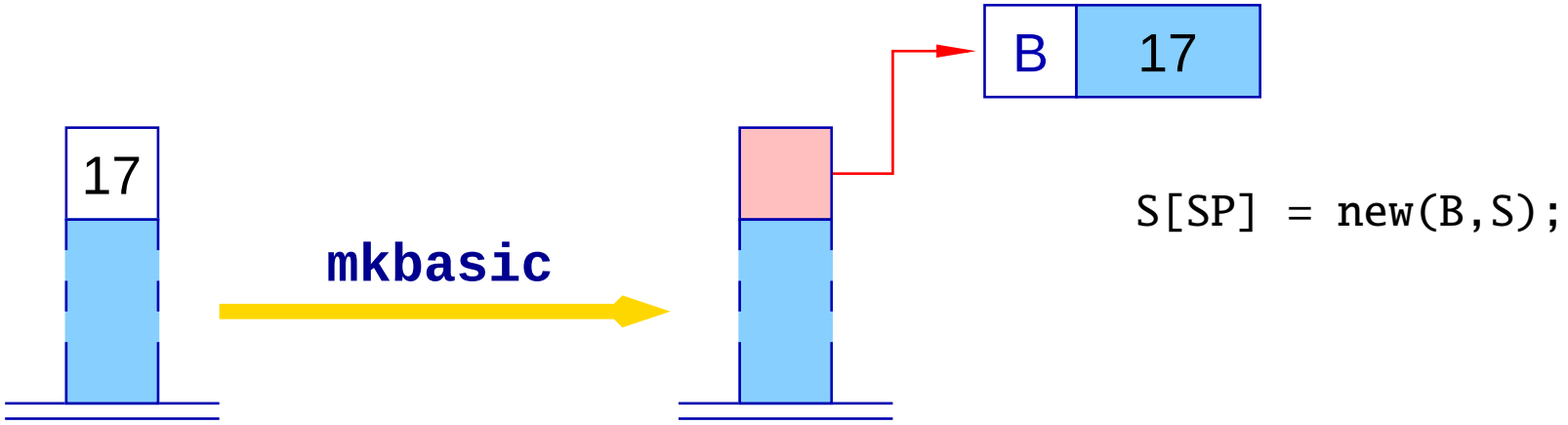
- Ekstraargument sd (stack difference) simuleerib registri SP liikumist kui käsud modifitseerivad magasini. Kasutame hiljem muutujate adresseerimisel.

Lihtsate avaldiste transleerimine

Funktsioon code_V lihtsate avaldiste jaoks on analoogne funktsiooniga code_B, kuid loob kuhja baasobjekti.

$$
\begin{array}{lcl}
\text{code}_V\ b\ \rho\ \text{sd} & = & \texttt{loadc b} \\
 & & \texttt{mkbasic} \\
\text{code}_V\ (\square_1\, e)\ \rho\ \text{sd} & = & \text{code}_B\ e\ \rho\ \text{sd} \\
 & & \text{op}_1 \\
 & & \texttt{mkbasic} \\
\text{code}_V\ (e_1\, \square_2\, e_2)\ \rho\ \text{sd} & = & \text{code}_B\ e_1\ \rho\ \text{sd} \\
 & & \text{code}_B\ e_2\ \rho\ (\text{sd}+1) \\
 & & \text{op}_2 \\
 & & \texttt{mkbasic}
\end{array}
$$

Lihtsate avaldiste transleerimine



$\text{code}_V\ (\textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2)\ \rho\ \text{sd} =$

$\text{code}_B\ e_0\ \rho\ \text{sd}$
`jumpz A`
$\text{code}_V\ e_1\ \rho\ \text{sd}$
`jump B`
A: $\text{code}_V\ e_2\ \rho\ \text{sd}$
B: $\ldots$

Ligipääs muutujatele

Näide: vaatame funktsiooni f

$$\begin{array}{lll} \textbf{let} & c = 5 & \\ & f = \textbf{fn}\ a \quad \Rightarrow & \textbf{let}\ b = a * a \\ & & \textbf{in}\ b + c \\ \textbf{in}\ \ldots & & \end{array}$$

Funktsioon f kasutab globaalset muutujat c ning lokaalseid muutujaid a (formaalne parameeter) ja b (defineeritud sisemise **let**-avaldise poolt).

Globaalse muutuja väärtus on määratud funktsiooni konstrueerimise ajal (staatiline skoopimine!) ning on täitmisajal otse kasutatav.

Ligipääs muutujatele

Globaalsed muutujad

- Globaalsete muutujatega seotud väärtused hoitakse kuhjas vektorina (Global Vector).
- Adresseeritakse üksteisele järgnevalt alates 0-st.
- F-objekti või C-objekti konstrueerimisel leitakse funktsiooni või avaldise globaalvektor ja tema aadress salvestatakse objekti gp-komponenti.
- Avaldise väärtustamise ajal viitab register GP (Global Pointer) hetkel kehtivale globaalvektorile.

Ligipääs muutujatele

Lokaalsed muutujad

Lokaalseid muutujaid hallatakse magasinis freimides.

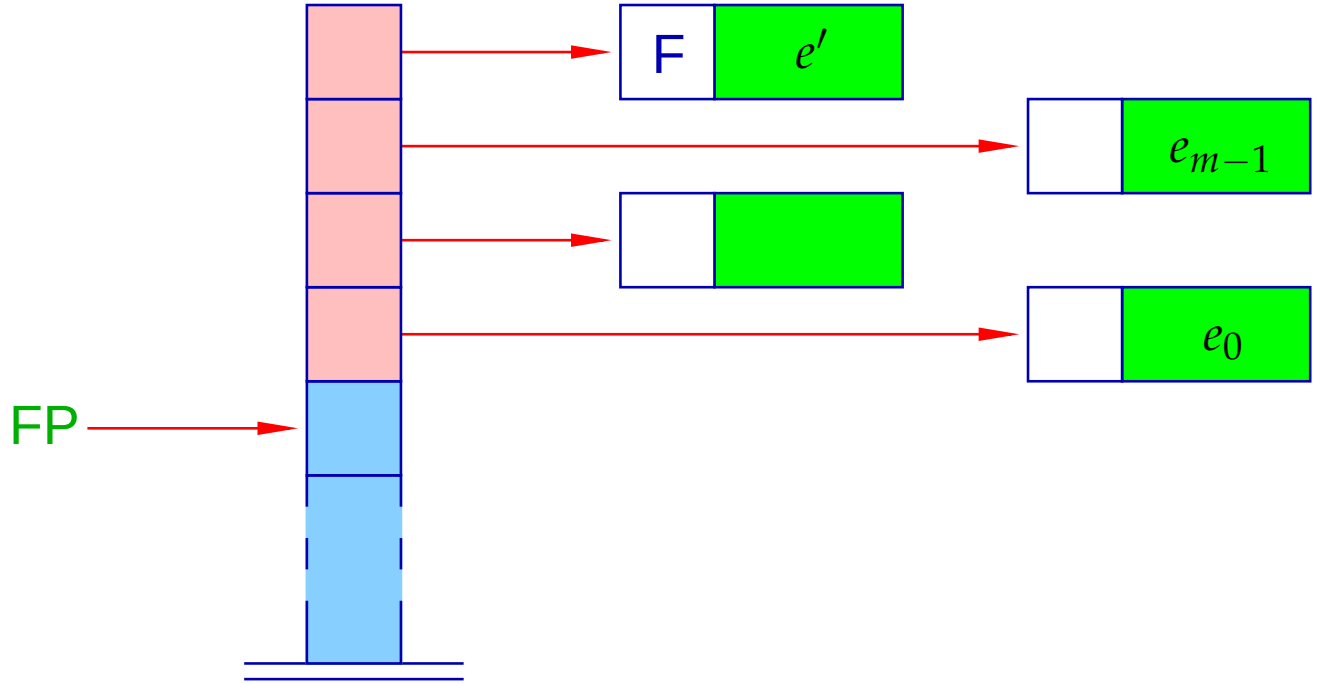
Olgu $e \equiv e'\ e_0\ \ldots\ e_{m-1}$ funktsiooni e' aplikatsioon argumentidele $e_0, \ldots, e_{m-1}$.

NB! Funktsiooni e' aarsus ei pea olema m.

- Lokaalseid muutujaid hallatakse magasinis freimides.
- PuF funktsioonid on karritud (curried) $f : t_1 \to t_2 \to \cdots \to t_n \to t$.
- Seega võib f olla rakendatud vähemale kui n argumendile.
- Kui t on funktsionalset tüüpi, siis võib f olla rakendatud ka rohkemale kui n argumendile.

Ligipääs muutujatele

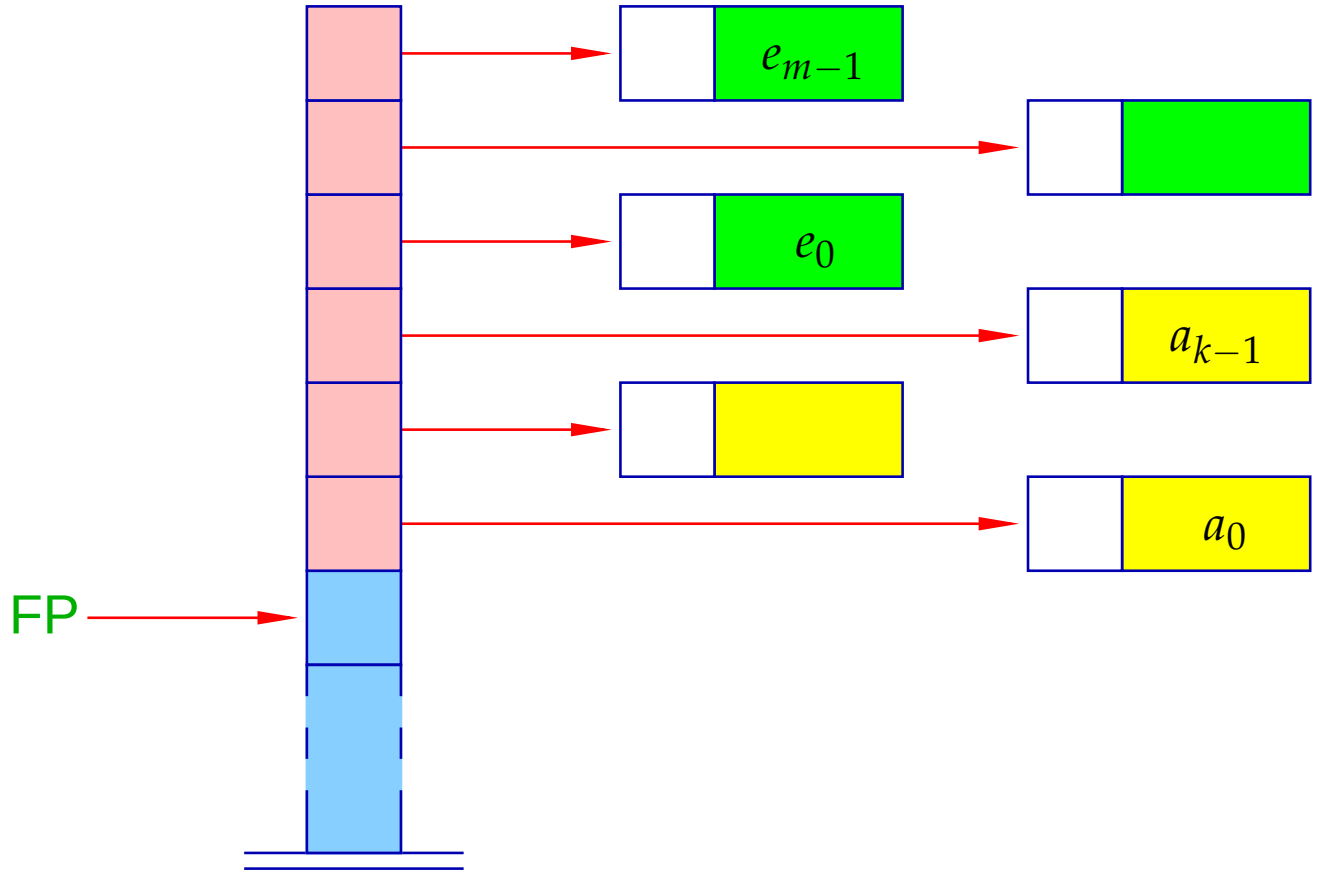
Võimalik magasini organiseerimine:



+ Parameetreid saab adresseerida FP suhtes.
– Avaldise e' lokaalseid muutujaid ei saa adresserida FP suhtes.
– Kui e' on n-aarne funktsioon ja $n < m$, siis ülejäänud $m - n$ argumenti tuleb freimis ümberpaigutada.

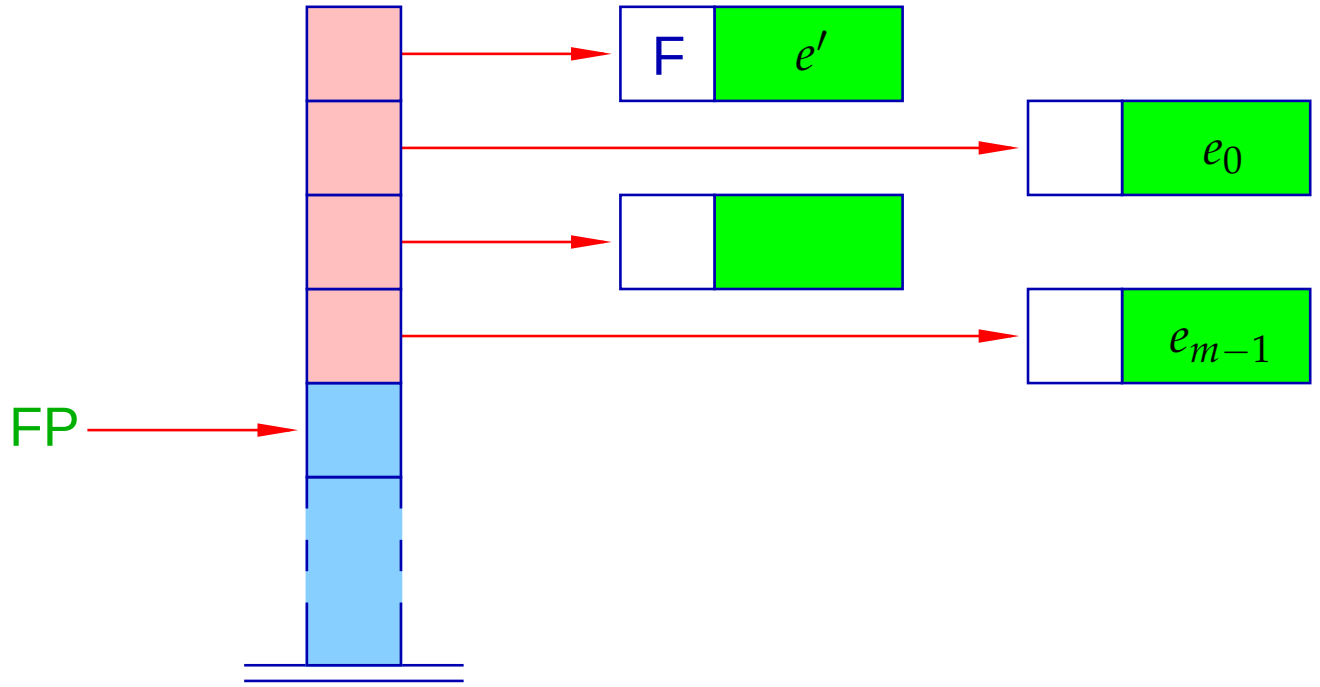
Ligipääs muutujatele

– Kui e' väärtustub funktsiooniks, mida on juba osaliselt rakendatud parameetritele $a_0, \ldots, a_{k-1}$, siis tuleb need liigutada e_0-i alla.



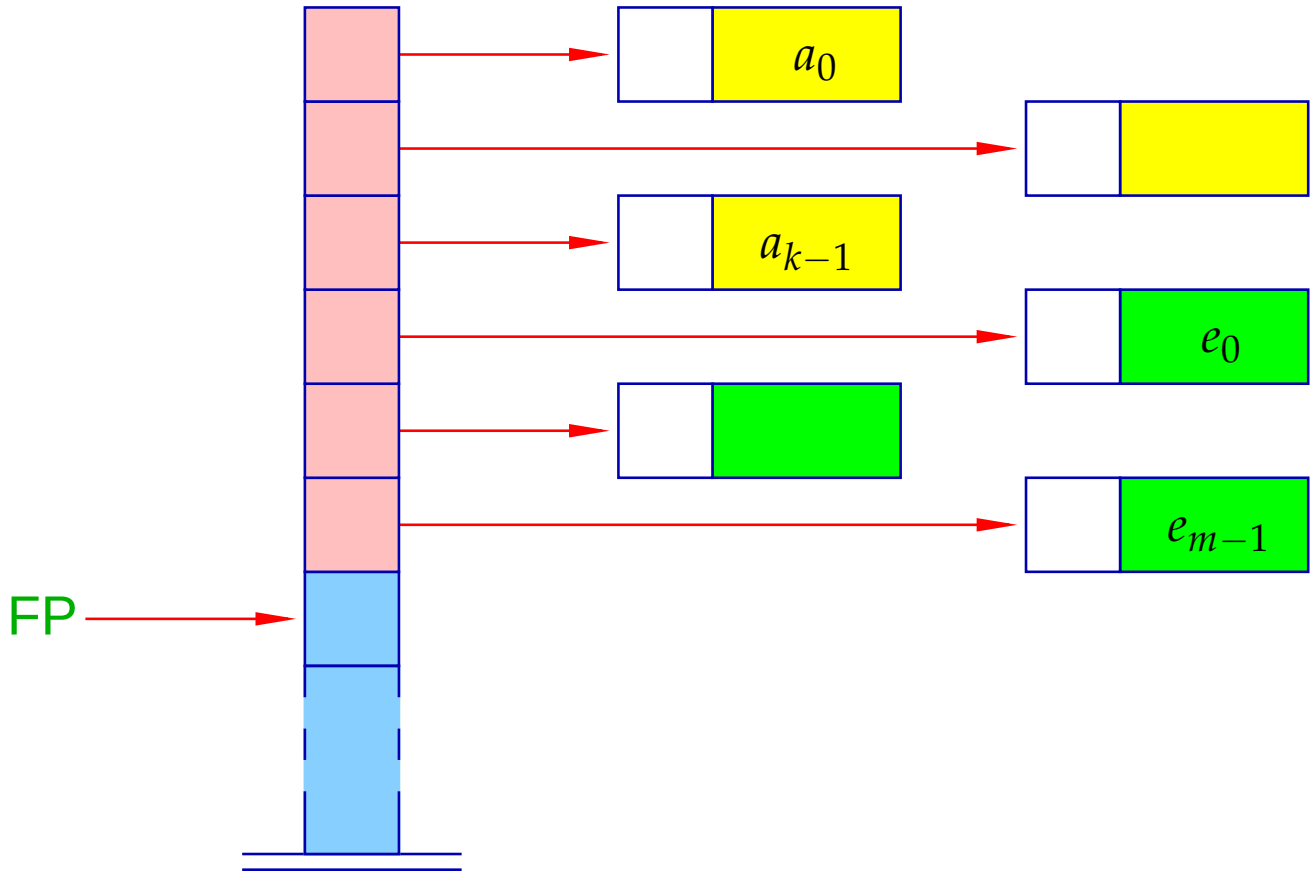
Ligipääs muutujatele

Alternatiivne magasini organiseerimine:



+ Lisaargumendid $a_0, \ldots, a_{k-1}$ ja lokaalsed muutujad saab salvestada magasini pärast argumente.

Ligipääs muutujatele

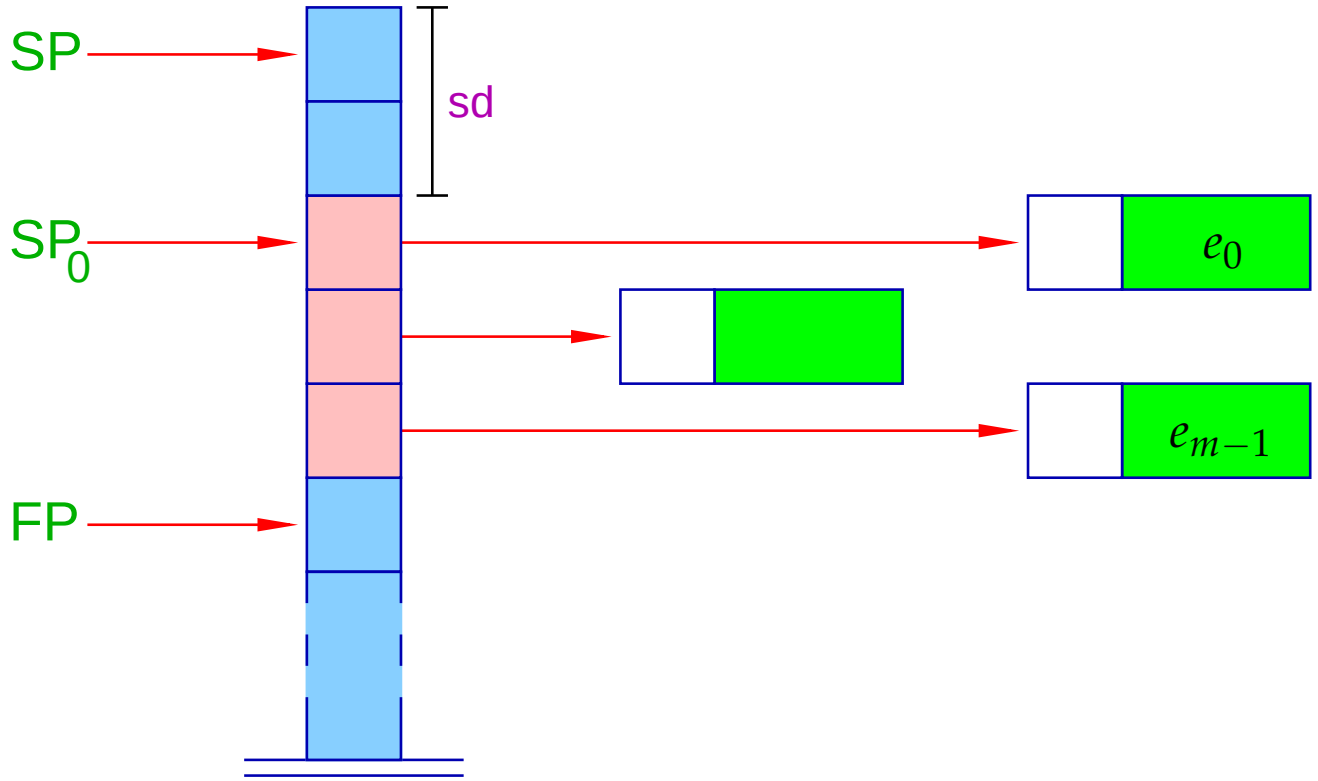


– Formaalsete parameetrite adresseerimine FP suhtes pole enam võimalik.

Ligipääs muutujatele

Lahendus:

- Adresseerime nii argumente, kui lokaalseid muutujaid, registri SP suhtes!
- Kuid SP muutub programmi täitmisajal . . .



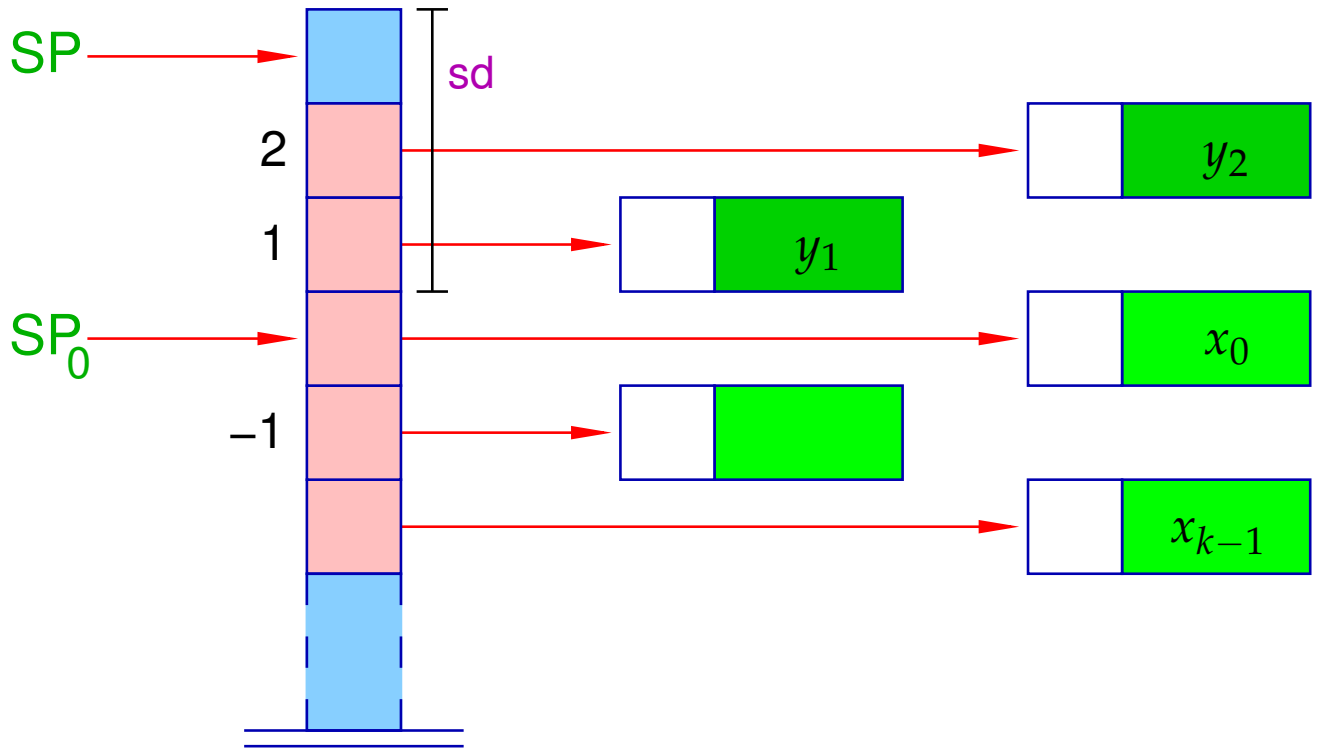
Ligipääs muutujatele

- Magasini kaugus, sd, näitab registri SP hetkeväärtuse erinevust tema väärtusest SP_0 funktsiooni kehasse sisenemisel.
- Magasini kaugust on võimalik määrata staatiliselt simuleerides magasini muutumist käskude täitmisel.
- Formaalsed parameetrid $x_0, x_1, x_2, \ldots$ seotakse mittepositiivsete suhtadressitega vastavalt 0, -1, -2, . . . ; so. $\rho\ x_i = (L, -i)$.
- i-nda formaalse parameetri absoluutaadress on:

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

- Lokaalsed **let**-muutujad lisatakse üksteisejärel magasini tippu.

Ligipääs muutujatele



- Lokaalsed muutujad $y_1, y_2, \ldots$ seotakse postiivsete suhtaadressitega; so. $\rho\ y_i = (L, i)$.
- i-nda lokaalse muutuja absoluutaadress on:

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

Ligipääs muutujatele

CBN semantika korral emiteeritakse muutujate väärtustamiseks kood:

$$\begin{array}{lcll} \text{code}_V\ x\ \rho\ \text{sd} & = & \texttt{pushloc}\ (\text{sd} - i) & \text{kui } \rho\ x = (L, i) \\ & & \texttt{eval} & \\ \text{code}_V\ x\ \rho\ \text{sd} & = & \texttt{pushglob i} & \text{kui } \rho\ x = (G, i) \\ & & \texttt{eval} & \end{array}$$

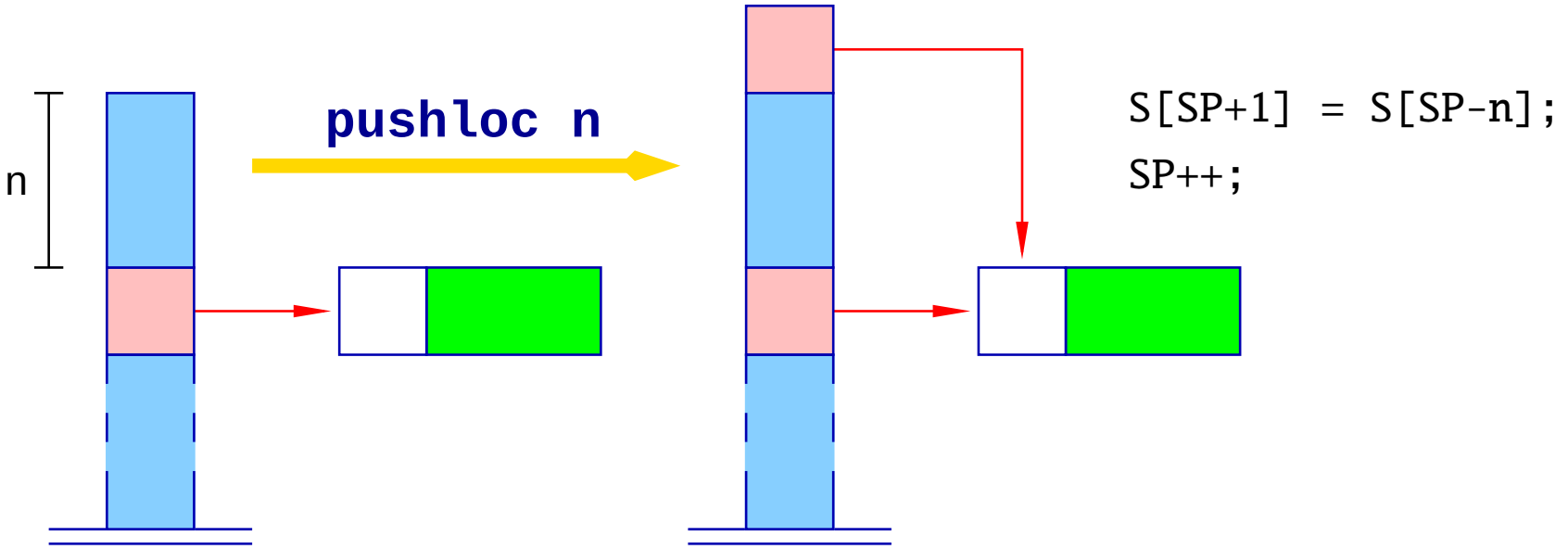
Käsk `eval` kontrollib, kas muutuja on juba väärtustatud või mitte ning teisel juhul väärtustab selle (vaatame lähemalt hiljem).

CBV semantika korral ei ole `eval` käsku vaja genereerida.

Ligipääs muutujatele

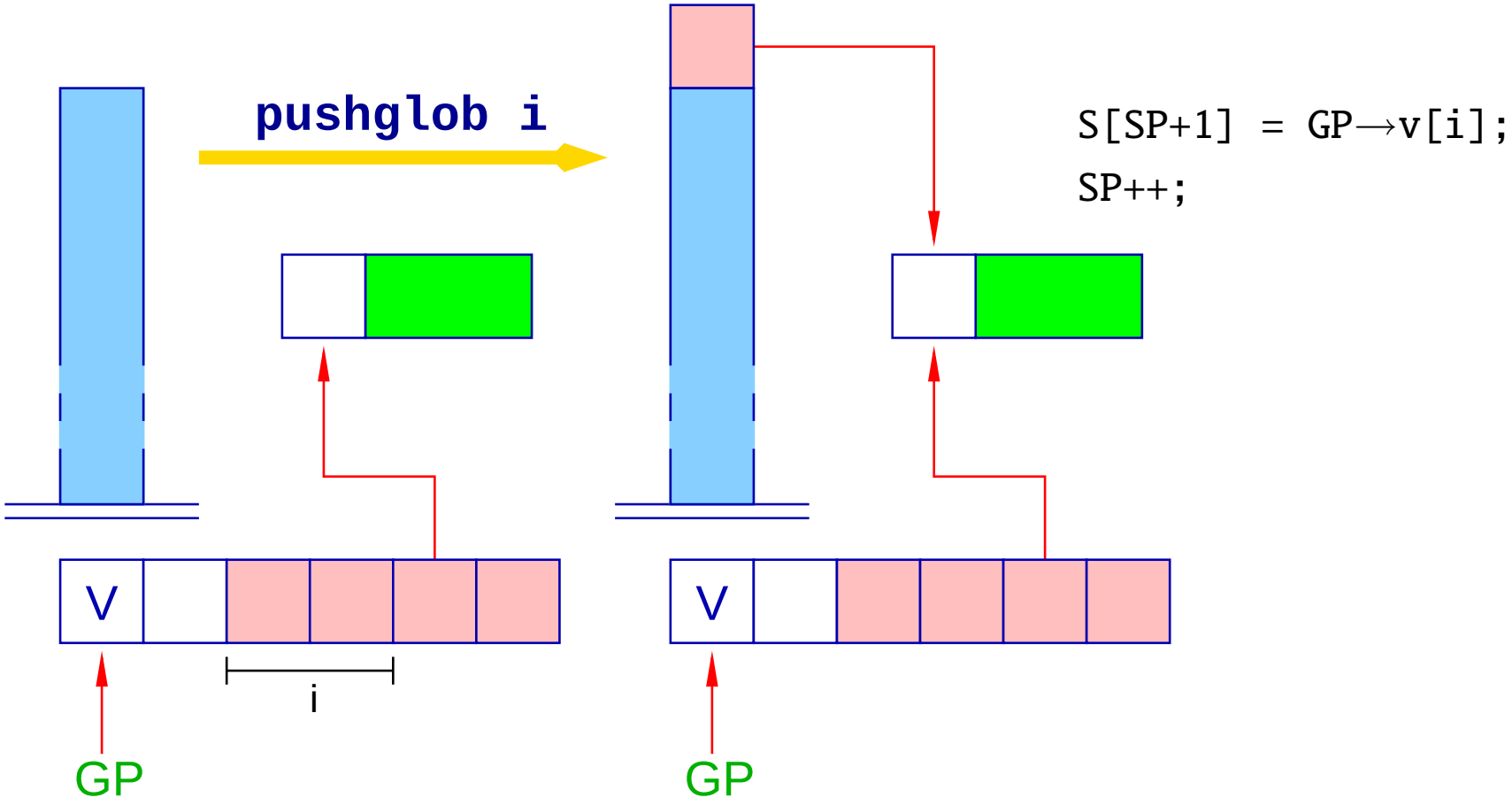
Lokaalne muutuja suhtaadressiga i loetakse magasinipesast $S[a]$, kus

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



Ligipääs muutujatele

Globaalne muutuja loetakse globaalvektorist.



Ligipääs muutujatele

Näide:

Olgu $e \equiv (b + c)$ keskkonnas $\rho = \{b \mapsto (L, 1), c \mapsto (G, 0)\}$ ja sd $= 1$.

CBN semantika korral $\text{code}_V\ e\ \rho$ sd emiteerib koodi:

```
1  pushloc 0        3  eval
2  eval             3  getbasic
2  getbasic         3  add
2  pushglob 0       2  mkbasic
```

Funktsioonidefinitsioonide transleerimine

Funktsiooni definitsiooni transleerimisel genereeritakse kood, mis loob kuhjas funktsionaalse väärtuse:

- luuakse globaalvektor globaalsete muutujate sidumiseks;
- luuakse (algselt tühi) argumentvektor;
- luuakse F-objekt, mis sisaldab viitu neile vektoreile ning funktsiooni kehale vastava koodi algusaadressile.

Eraldi genereeritakse funktsiooni definitsiooni kehale vastav kood.

Funktsioonidefinitsioonide transleerimine

$\text{code}_V\ (\textbf{fn}\ x_0, \ldots, x_{k-1} \Rightarrow e)\ \rho\ \text{sd}\quad =$

getvar $z_0\ \rho\ \text{sd}$	`mkvec g`	A: `targ k`
getvar $z_1\ \rho\ (\text{sd}+1)$	`mkfunval A`	$\text{code}_V\ e\ \rho'\ 0$
$\ldots$	`jump B`	`return k`
getvar $z_{g-1}\ \rho\ (\text{sd}+g-1)$		B: $\ldots$

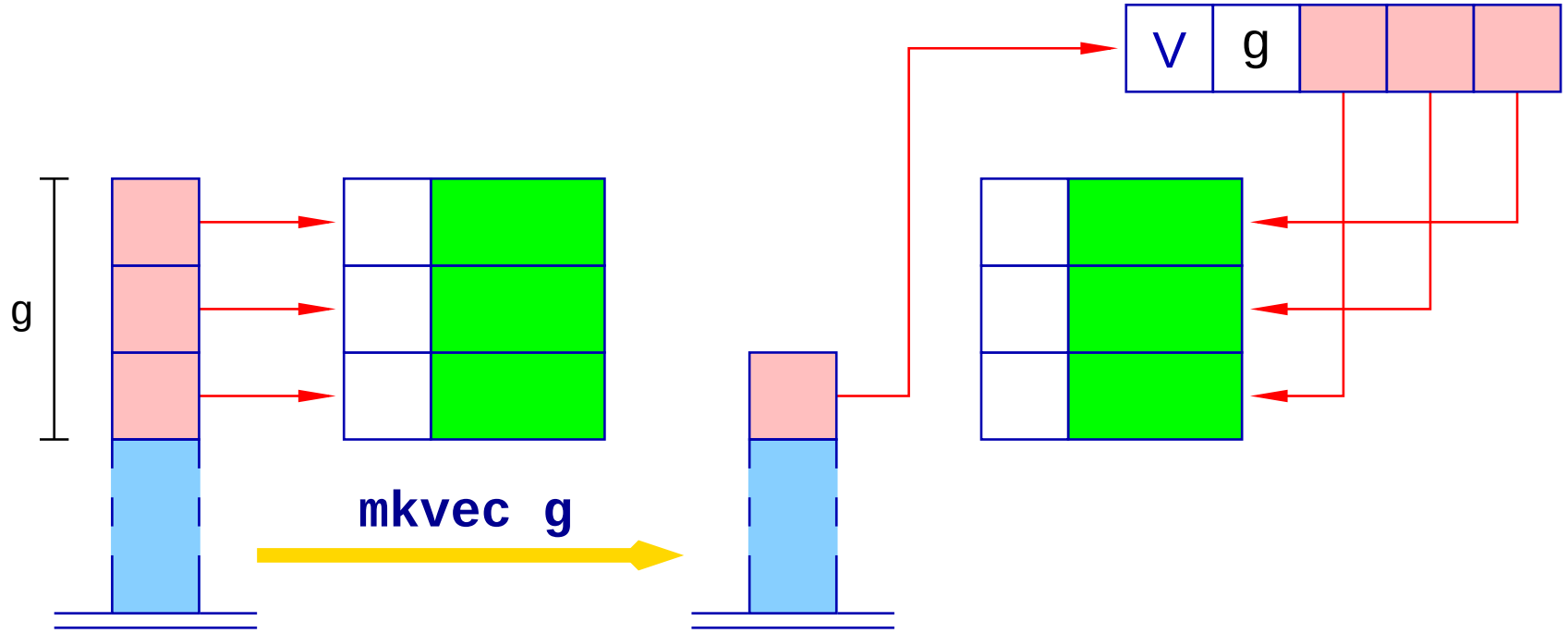
$$\textbf{kus}\quad \{z_0, \ldots, z_{g-1}\} = \mathit{free}(\textbf{fn}\ x_0, \ldots, x_{k-1} \Rightarrow e)$$

$$\rho' = \{x_i \mapsto (L, -i) \mid i = 0, \ldots, k-1\}$$

$$\cup\ \{z_j \mapsto (G, j) \mid j = 0, \ldots, g-1\}$$

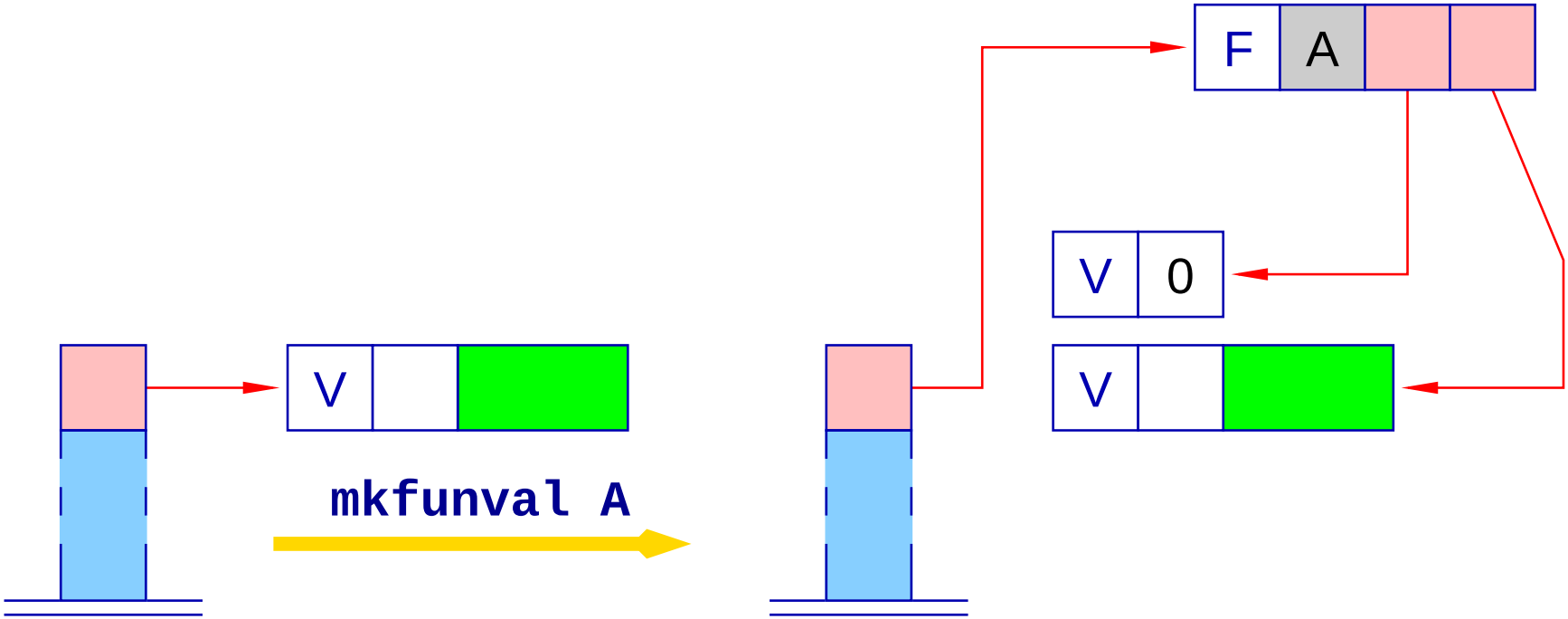
$$\text{getvar}\ y\ \rho\ \text{sd} = \begin{cases} \texttt{pushloc}\ (\text{sd}-i) & \text{kui}\ \rho\ y = (L, i) \\ \texttt{pushglob}\ \texttt{j} & \text{kui}\ \rho\ y = (G, j) \end{cases}$$

Funktsioonidefinitsioonide transleerimine




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

Funktsioonidefinitsioonide transleerimine




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

Funktsioonidefinitsioonide transleerimine

Näide: olgu $f \equiv \textbf{fn}\ b \Rightarrow a + b$ keskkonnas $\rho = \{a \mapsto (L, 1)\}$ ja sd $= 2$.

$\text{code}_V\ f\ \rho$ sd emiteerib koodi:

```
2      pushloc 1        0  pushglob 0        2      getbasic
3      mkvec 1          1  eval              2      add
3      mkfunval A       1  getbasic          1      mkbasic
3      jump B           1  pushloc 1         1      return 1
0  A:  targ 1           2  eval              3  B:  ...
```

Käske `targ k` ja `return k` vaatame hiljem.

Aplikatsioonide transleerimine

Funktsiooni aplikatsiooni $e'\ e_0\ \ldots\ e_{m-1}$ korral genereeritakse kood, mis:

- loob magasinis uue freimi;
- kannab üle aktuaalsed parameetrid; so.

  CBV: väärtustab aktuaalsed parameetrid;

  CBN: loob aktuaalsete parameetrite sulundid;
- väärtustab funktsiooni e' F-objektiks;
- rakendab funktsiooni argumentidele.

Aplikatsioonide transleerimine

CBN korral genereeritakse kood:

$$
\begin{aligned}
\text{code}_V\ (e'\ e_0\ \ldots\ e_{m-1})\ \rho\ \text{sd} \quad = \quad & \texttt{mark A} \\
& \text{code}_C\ e_{m-1}\ \rho\ (\text{sd}+3) \\
& \text{code}_C\ e_{m-2}\ \rho\ (\text{sd}+4) \\
& \ldots \\
& \text{code}_C\ e_0\ \rho\ (\text{sd}+m+2) \\
& \text{code}_V\ e'\ \rho\ (\text{sd}+m+3) \\
& \texttt{apply} \\
\text{A:} \quad & \ldots
\end{aligned}
$$

CBV korral on argumentide e_i jaoks code_C asemel code_V.

Aplikatsioonide transleerimine

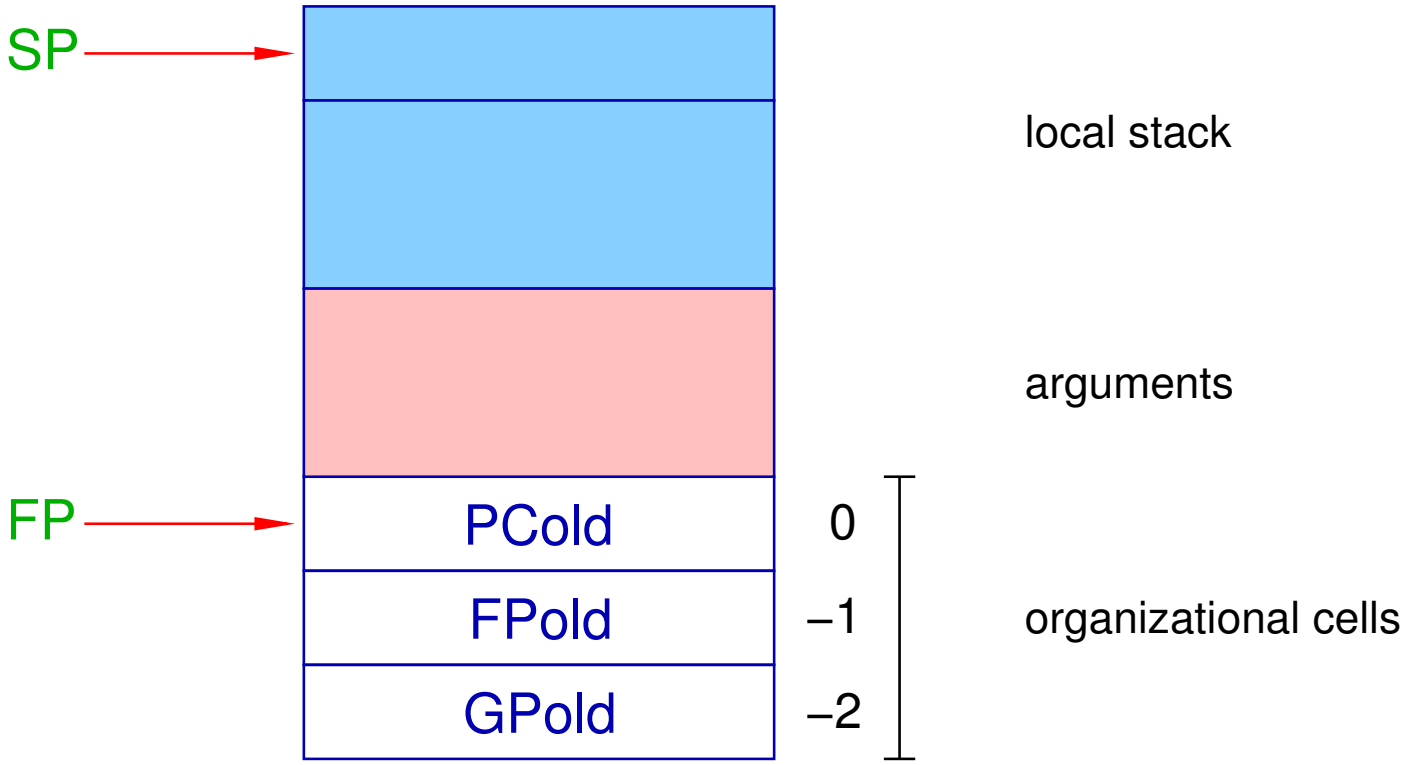
Näide: olgu $e \equiv f\ 42$ keskkonnas $\rho = \{f \mapsto (L, 2)\}$ ja sd $= 2$.

$\text{code}_V\ e\ \rho$ sd emiteerib CBV korral koodi:

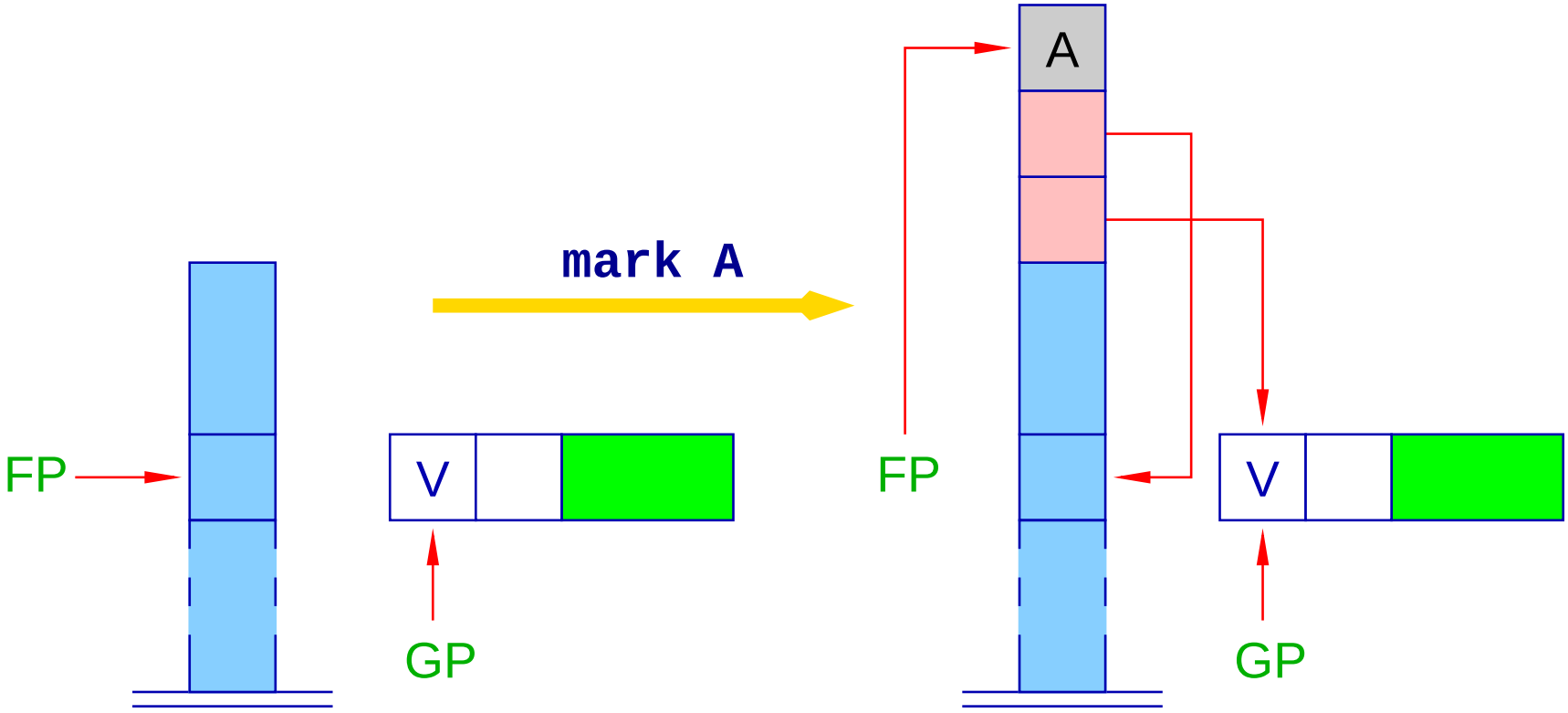
```
2   mark A          6       pushloc 4
5   loadc 42        7       apply
6   mkbasic         3   A:  ...
```

Aplikatsioonide transleerimine

Freimi struktuur:

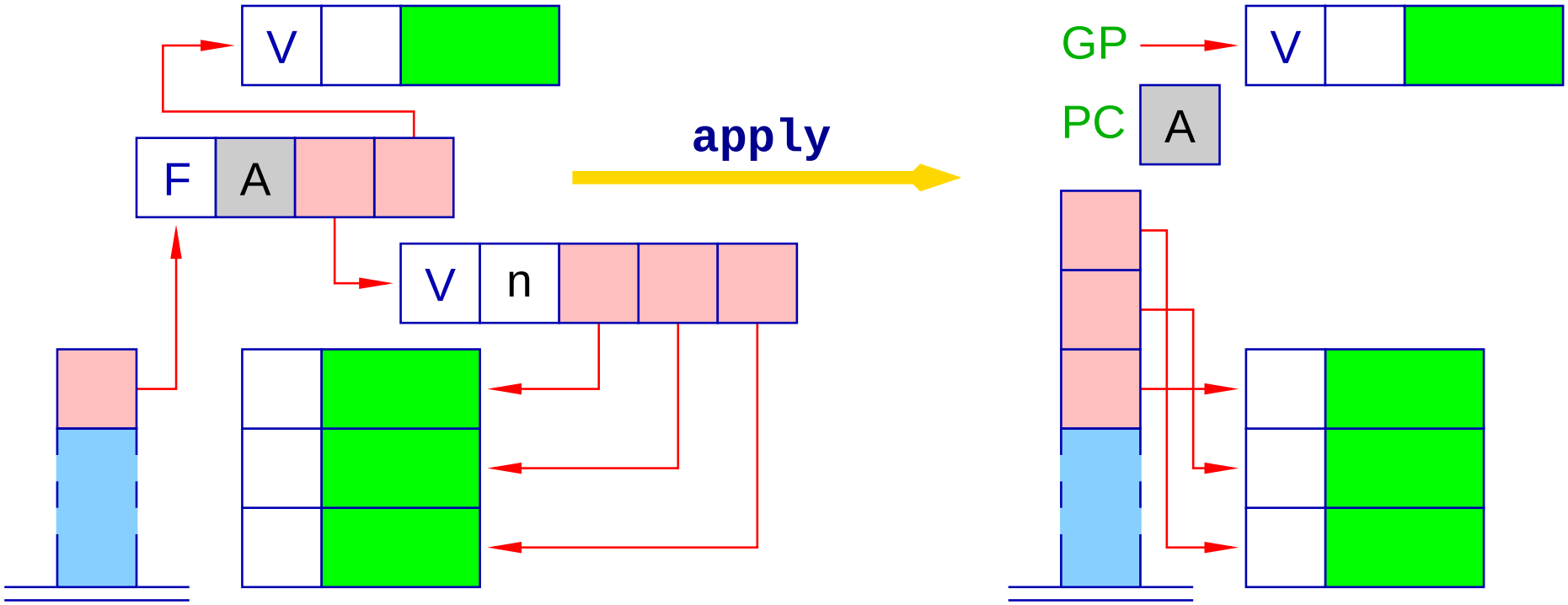


Aplikatsioonide transleerimine




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

Aplikatsioonide transleerimine



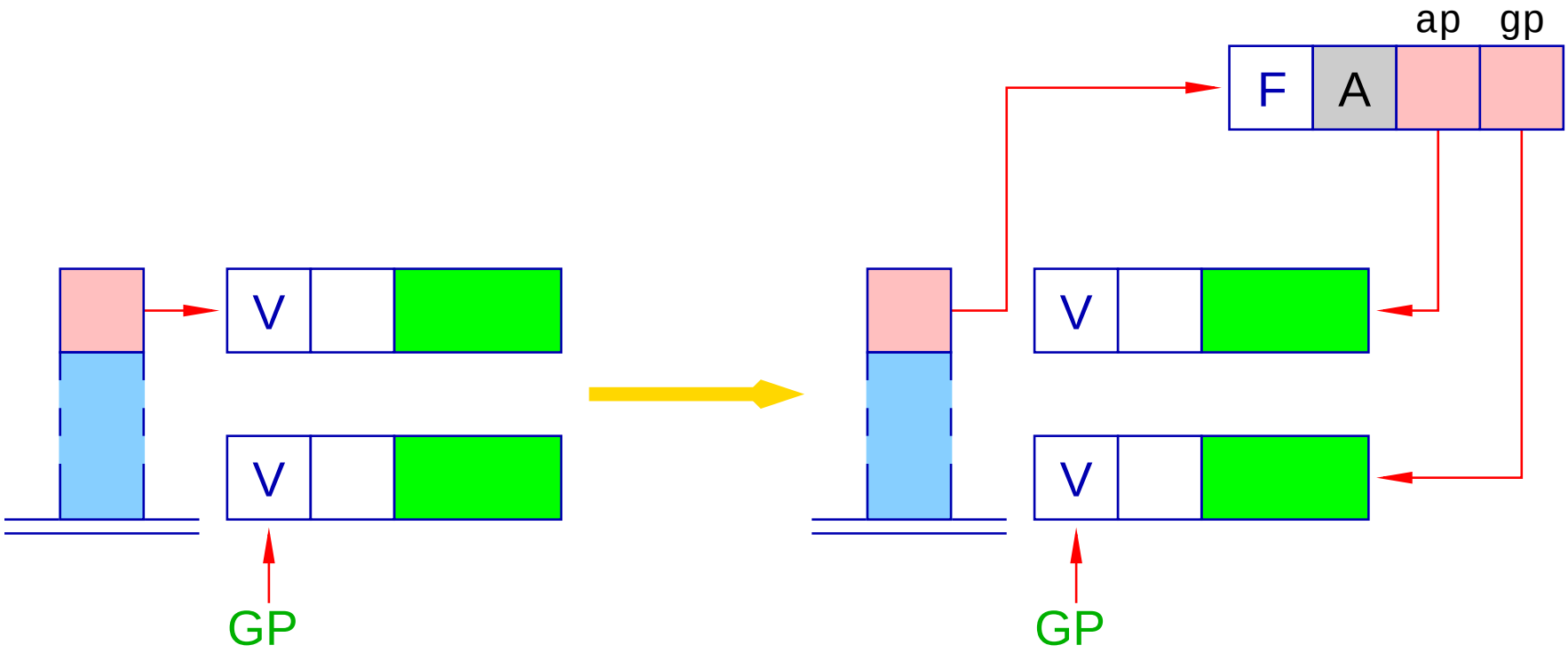

```
h = S[SP];
if (h→tag ≠ F)
  Error("Not Function");
else {
                                GP = h→gp; PC = h→cp;
                                for (i=0; i < h→ap→n; i++)
                                  S[SP+i] = h→ap→v[i];
                                SP = SP + h→ap→n - 1;
                              }
```

Argumentidega ala- ja ülevarustatus

- Esimene käsk pärast käsku `apply` on `targ k`.
- Kontrollib, et oleks funktsiooni väärtustamiseks piisavalt argumente
  - kontrollimiseks kasutab tingimust $\mathrm{SP} - \mathrm{FP} \geq k$.
- Kui on, siis alustatakse funktsiooni kehale vastava koodi täitmist.
- Vastasel korral väljastatakse uus funktsionaalne väärtus:
  - luuakse argumentvektor;
  - luuakse uus F-objekt;
  - magasinis vabastatakse freim.

Argumentidega ala- ja ülevarustatus

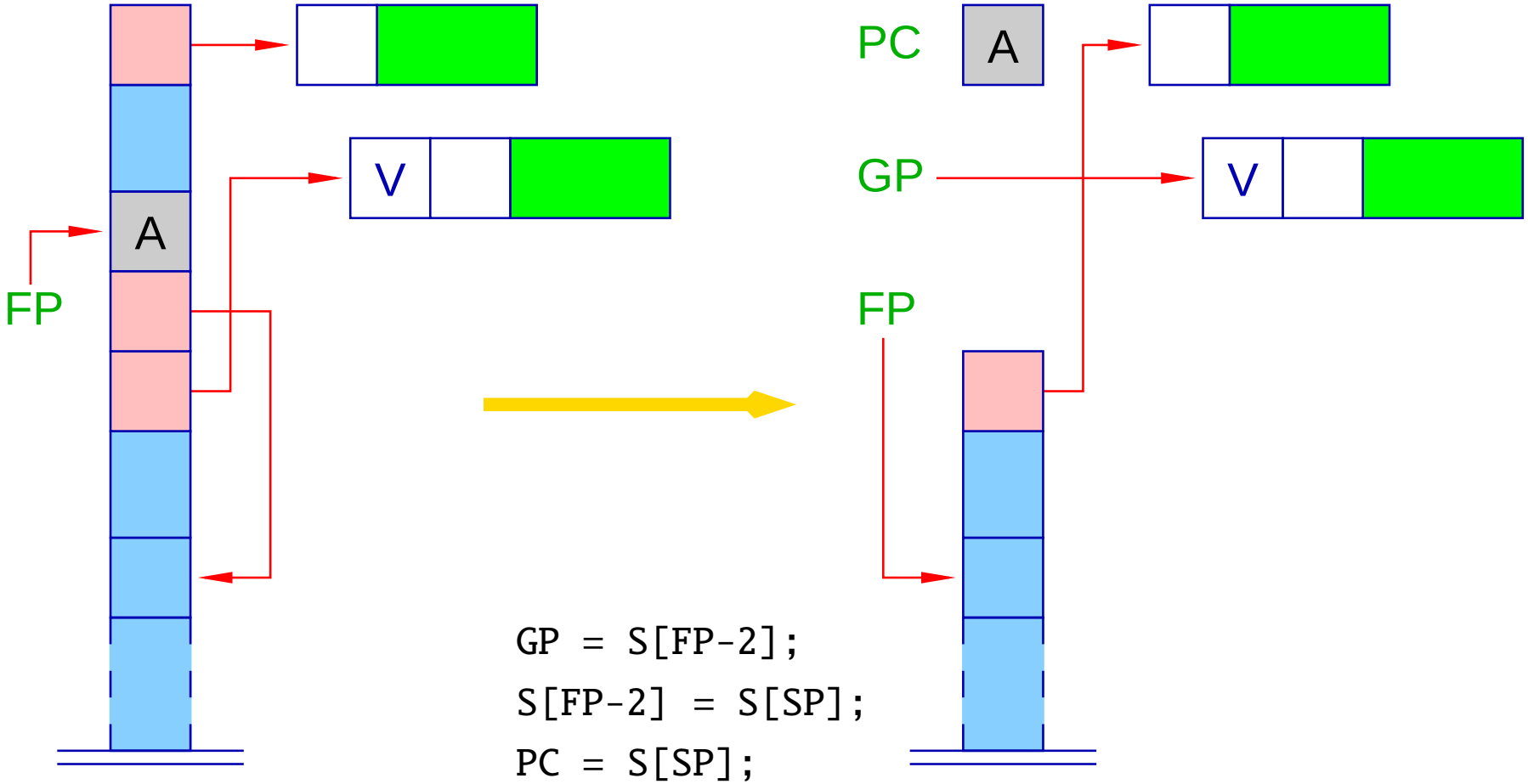
F-objekti konstrueerimine:




```
S[SP] = new(F,A,S[SP],GP);
```

Argumentidega ala- ja ülevarustatus

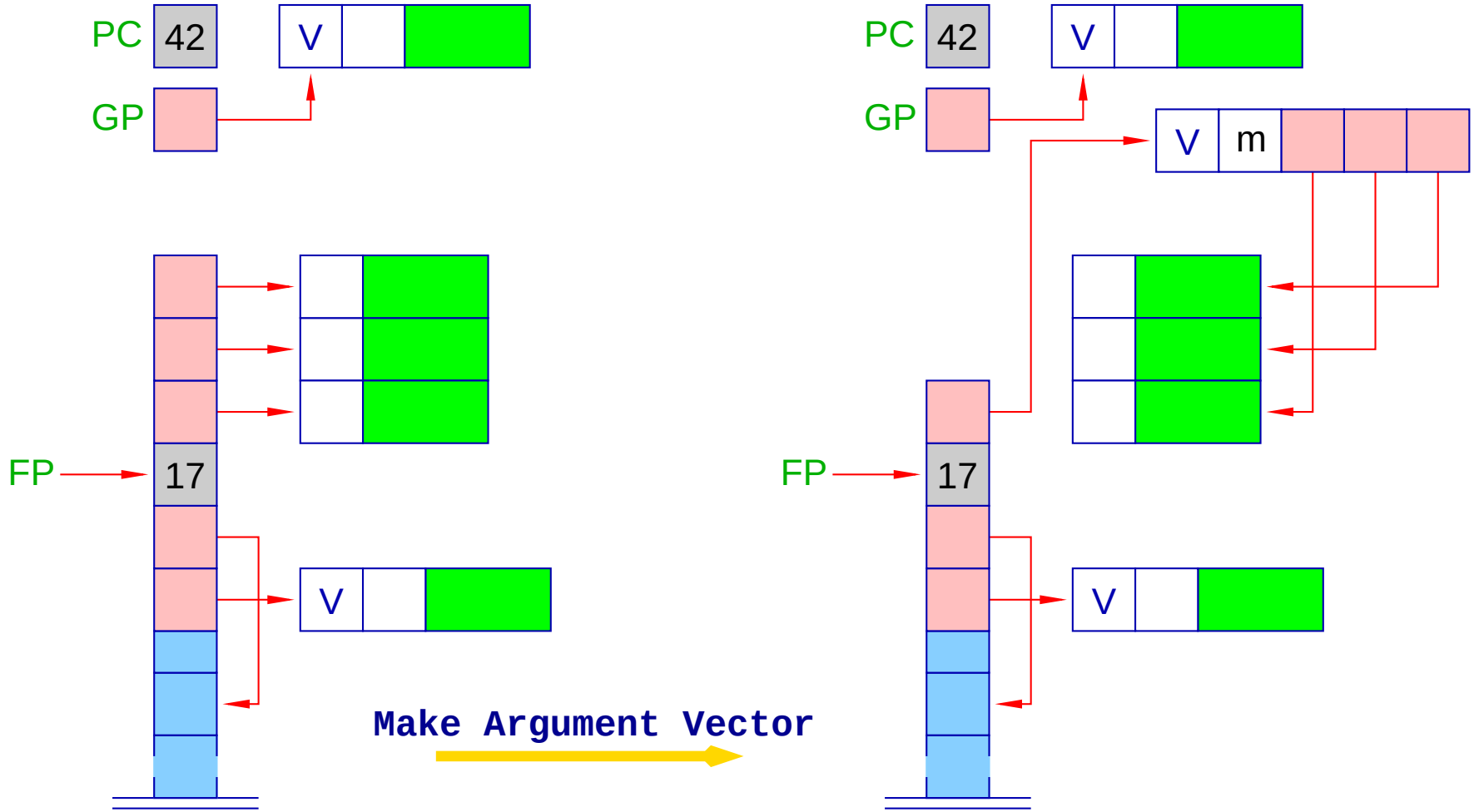
Freimi vabastamine:




```
GP = S[FP-2];
S[FP-2] = S[SP];
PC = S[SP];
SP = FP-2;
FP = S[FP-1];
```

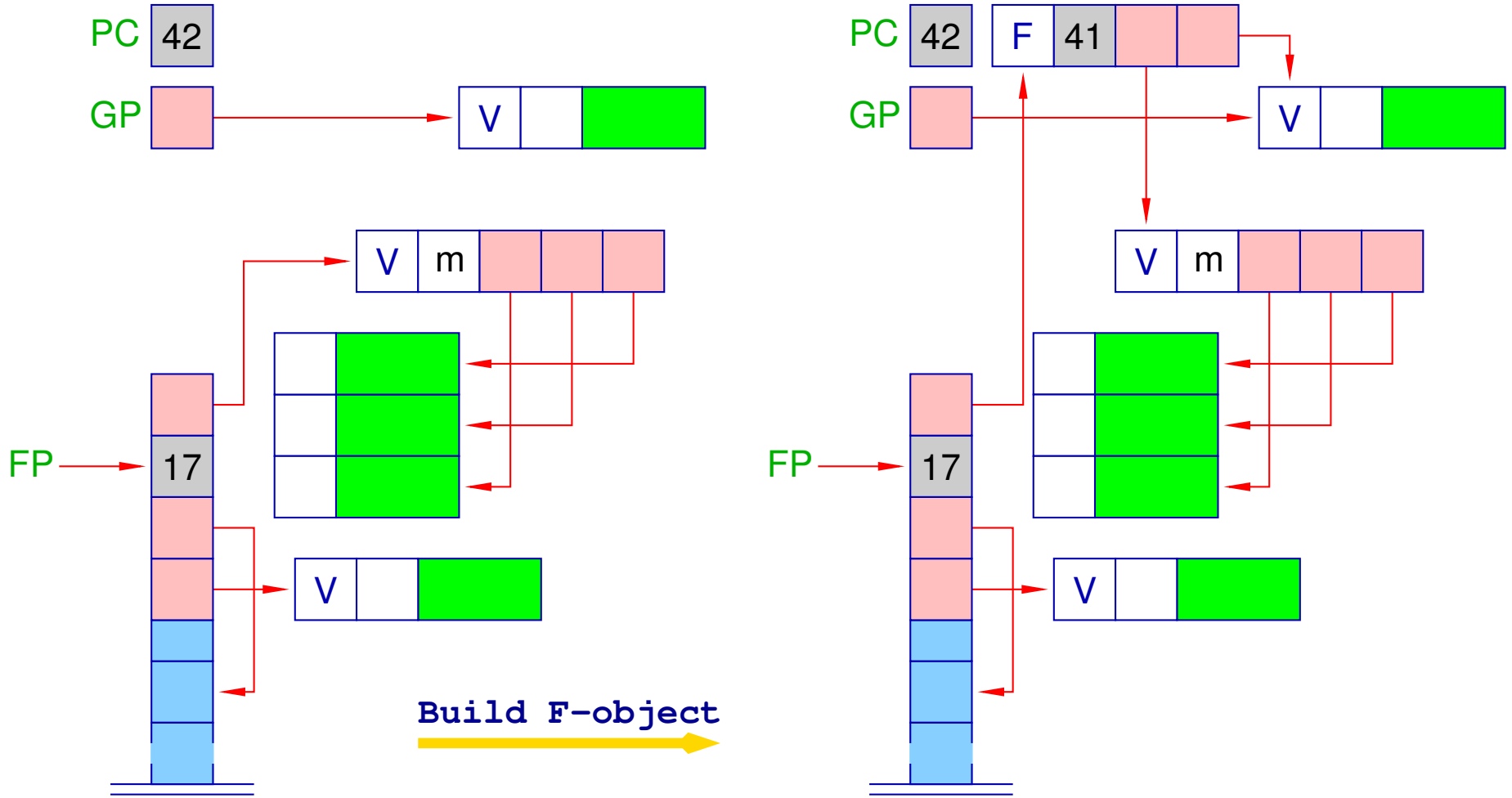
Argumentidega ala- ja ülevarustatus

`targ k`, kui argumente on $m < k$



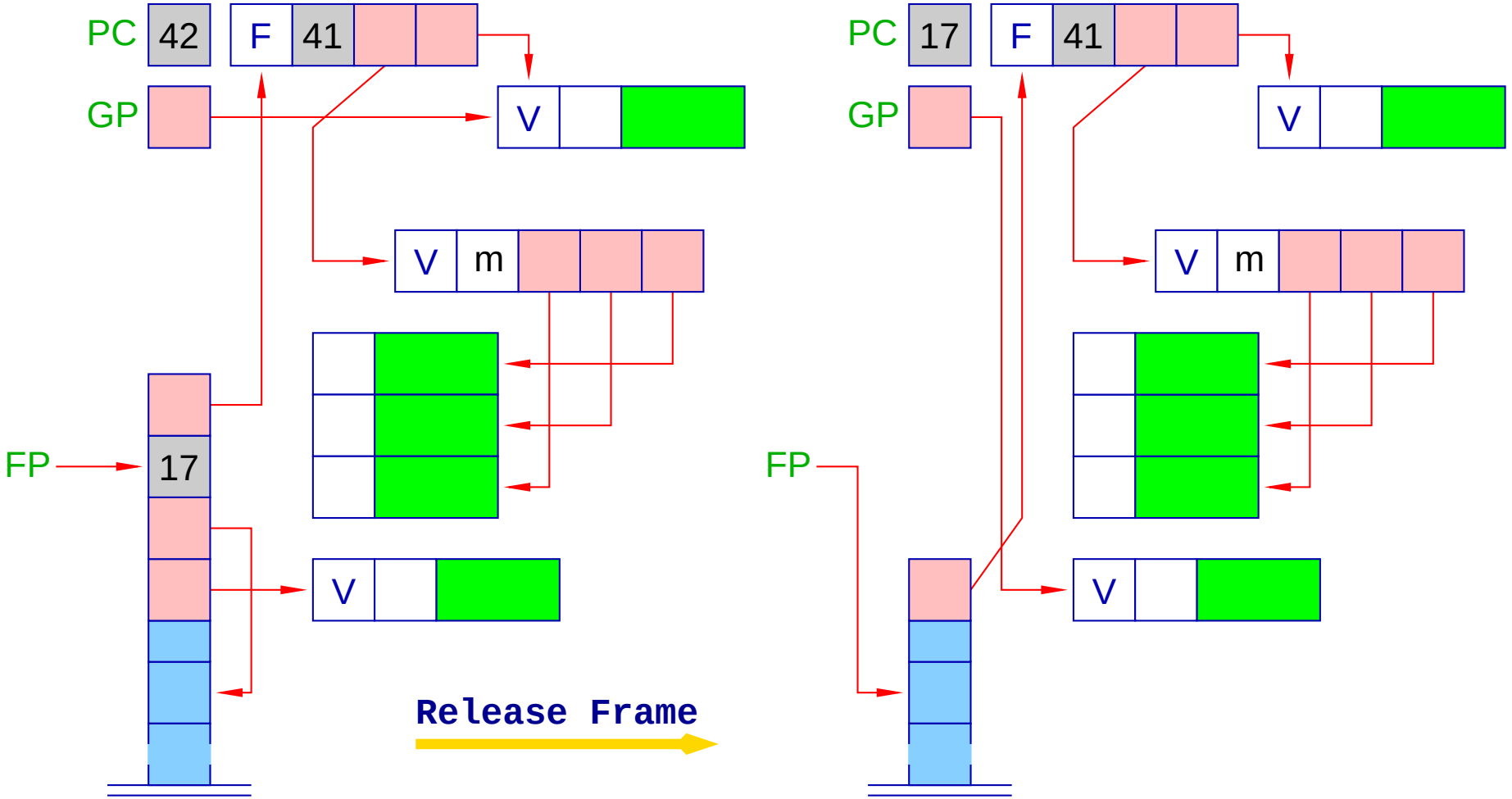
Argumentidega ala- ja ülevarustatus

`targ k`, kui argumente on $m < k$



Argumentidega ala- ja ülevarustatus

`targ k`, kui argumente on $m < k$



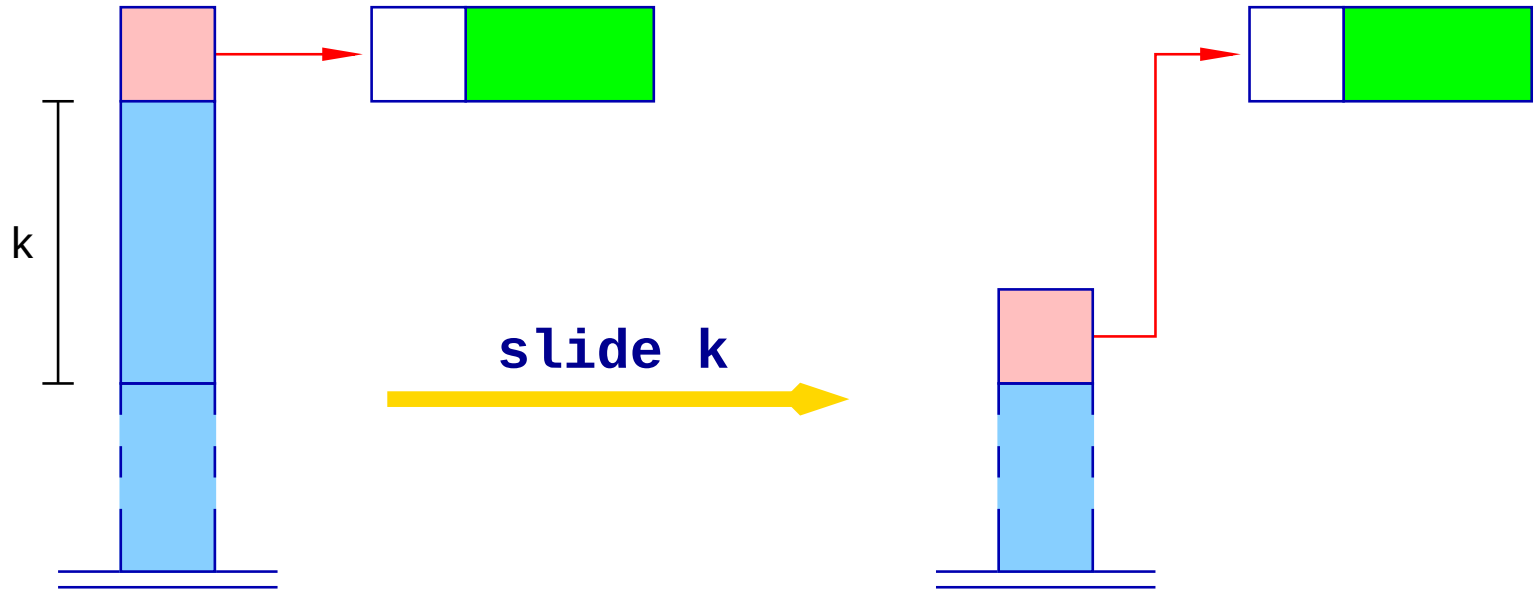
Argumentidega ala- ja ülevarustatus

- Funktsiooni keha kõige viimane käsk `return k` kontrollib, kas argumente oli tapselt õige arv.
- Kui oli, siis freim magasinis vabastatakse.
- Vastasel korral pidi funktsioon väärtustuma uueks funktsiooniks, mis kasutab ära ülejäänud argumendid.

```
return k  =  if (SP-FP = k+1)
               Release Stack Frame;
             else {
               slide k;
               apply;
             }
```

Argumentidega ala- ja ülevarustatus

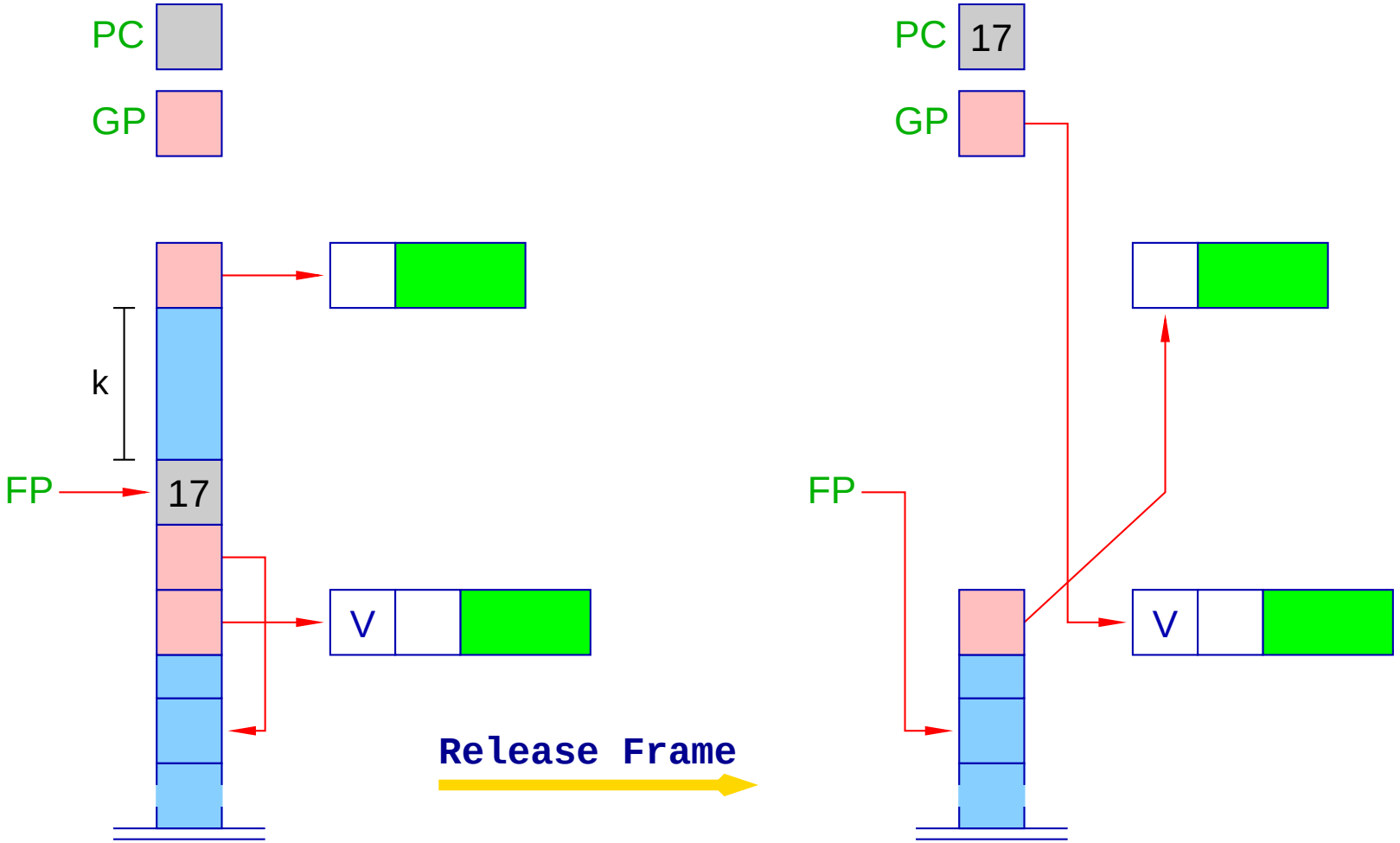
Käsk `slide k` nihutab magasini tipu k pesa allapoole, eemaldades vahepealsed pesad:




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

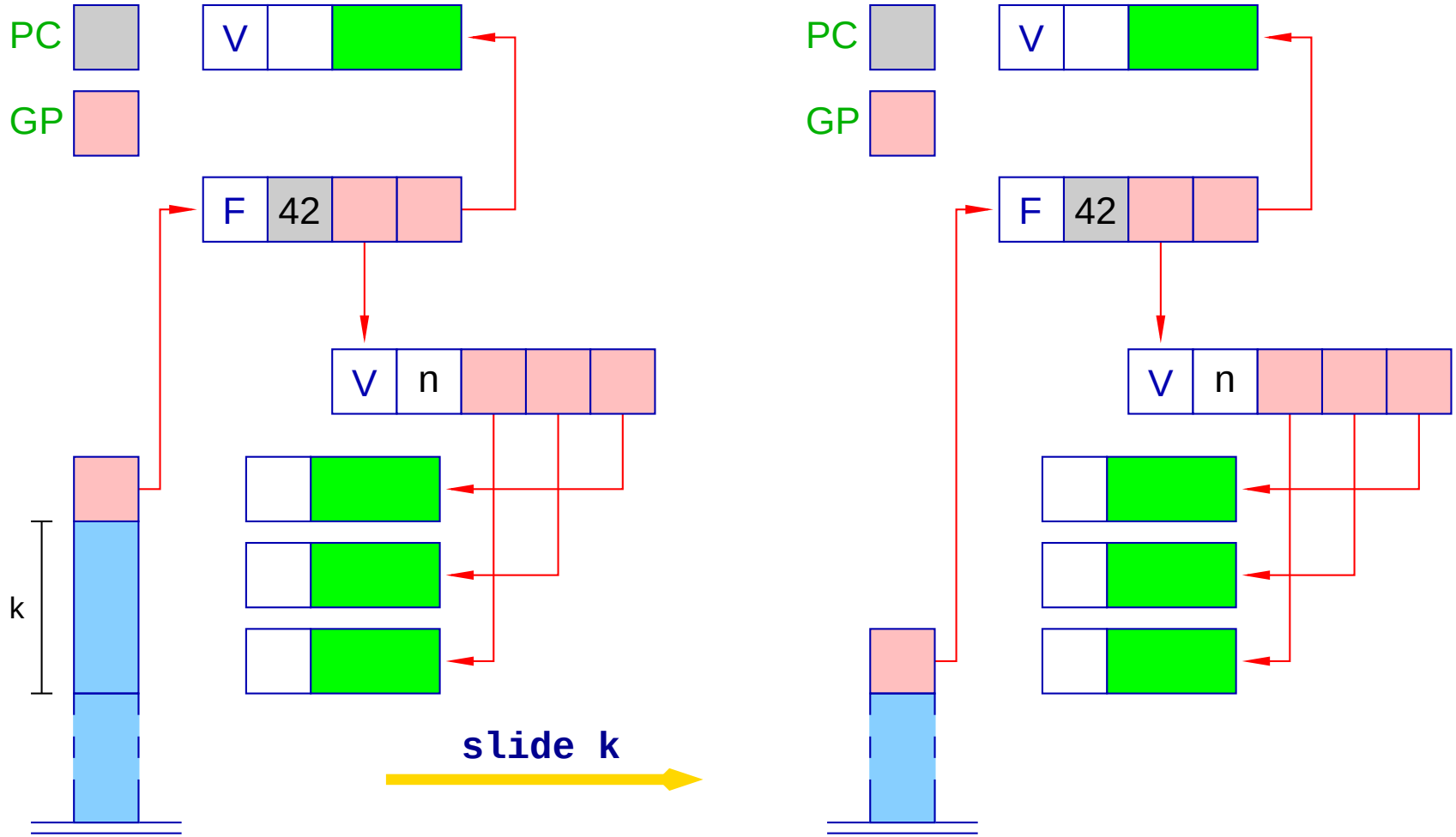
Argumentidega ala- ja ülevarustatus

`return k`, kui argumente on k



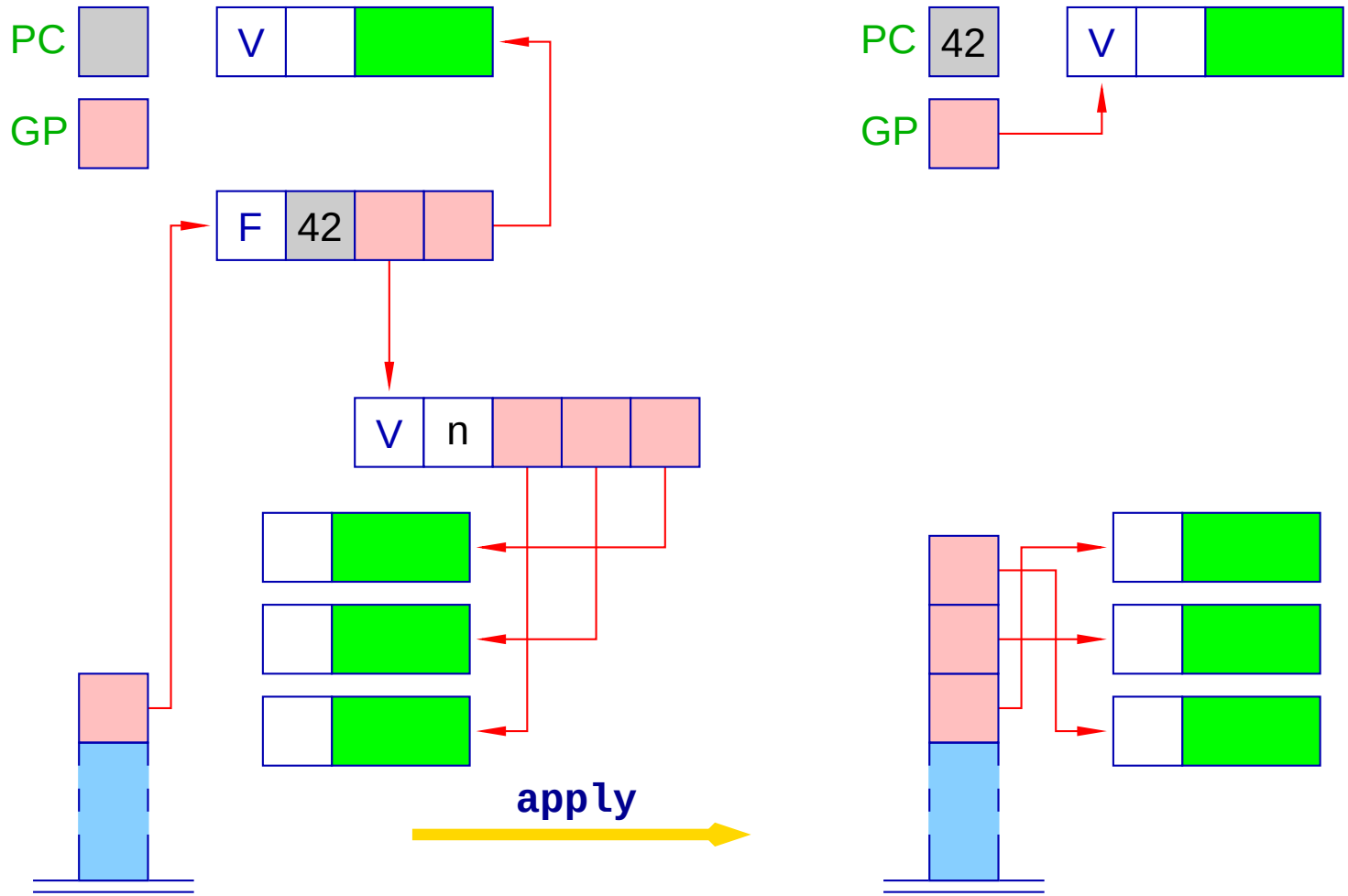
Argumentidega ala- ja ülevarustatus

`return k`, kui argumente on $m > k$



Argumentidega ala- ja ülevarustatus

`return k`, kui argumente on $m > k$



Lokaalsete definitsioonide transleerimine

Let-avaldise **let** $y_1 = e_1;\ \ldots;\ y_n = e_n$ **in** e_0 korral genereeritakse kood, mis:

- seob muutujad $y_1, \ldots, y_n$ vastavate väärtustega; so.

  CBV: väärtustab avaldised $e_1, \ldots, e_n$ ja seob muutujad nendega;

  CBN: seob muutujad avaldiste $e_1, \ldots, e_n$ sulunditega;

- väärtustab avaldise e_0 ja väljastab selle väärtuse.

Letrec-avaldises **letrec** $y_1 = e_1;\ \ldots;\ y_n = e_n$ **in** e_0 võivad avaldised e_i kasutada muutujaid y_j enne vastavate seoste loomist:

- muutujad seotakse algul fiktiivsete väärtustega, mis hiljem muudetakse ära.

Lokaalsete definitsioonide transleerimine

CBN korral genereeritakse kood:

$$\begin{aligned}
&\text{code}_V\ (\textbf{let}\ y_1 = e_1;\ \ldots;\ y_n = e_n\ \textbf{in}\ e_0)\ \rho\ \text{sd} \quad = \\
&\qquad\qquad \text{code}_C\ e_1\ \rho\ \text{sd} \\
&\qquad\qquad \text{code}_C\ e_2\ \rho_1\ (\text{sd}+1) \\
&\qquad\qquad \ldots \\
&\qquad\qquad \text{code}_C\ e_n\ \rho_{n-1}\ (\text{sd}+n-1) \\
&\qquad\qquad \text{code}_V\ e_0\ \rho_n\ (\text{sd}+n) \\
&\qquad\qquad \texttt{slide n}
\end{aligned}$$

kus $\rho_i = \rho \oplus \{y_j \mapsto (L, \text{sd}+j) \mid j = 1, \ldots, i\}$.

CBV korral on avaldiste e_i jaoks code_C asemel code_V.

NB! Kõigi avaldistega e_i on seotud sama globaalkeskkond.

Lokaalsete definitsioonide transleerimine

Näide: olgu $e \equiv \textbf{let}\ a = 19;\ b = a * a\ \textbf{in}\ a + b$ keskkonnas $\rho = \emptyset$.

$\text{code}_V\ e\ \rho\ 0$ emiteerib CBV korral koodi:

```
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       2  getbasic        3  slide 2
```

Lokaalsete definitsioonide transleerimine

CBN korral genereeritakse kood:

$$
\begin{array}{lll}
\text{code}_V\ (\textbf{letrec}\ y_1 = e_1;\ \ldots;\ y_n = e_n\ \textbf{in}\ e_0)\ \rho\ \text{sd} & = & \texttt{alloc n} \\
 & & \text{code}_C\ e_1\ \rho'\ (\text{sd}+n) \\
 & & \texttt{rewrite n} \\
 & & \ldots \\
 & & \text{code}_C\ e_n\ \rho'\ (\text{sd}+n) \\
 & & \texttt{rewrite 1} \\
 & & \text{code}_V\ e_0\ \rho'\ (\text{sd}+n) \\
 & & \texttt{slide n}
\end{array}
$$

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

CBV korral on avaldiste e_i jaoks code_C asemel code_V.

NB! Avaldised e_i ei tohi CBV korral olla baasväärtused.

Lokaalsete definitsioonide transleerimine

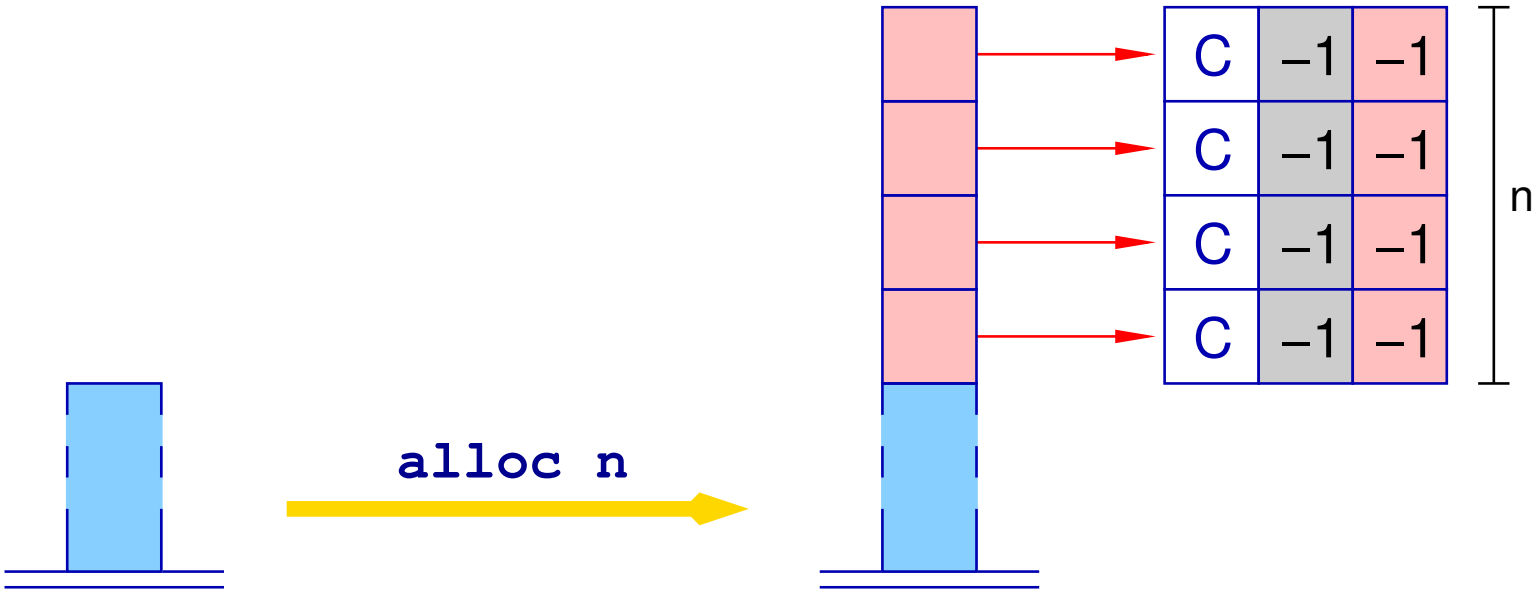
Näide:

$$e \equiv \textbf{letrec } f = \textbf{fn } x,y \Rightarrow \textbf{if } y \leq 1 \textbf{ then } x \textbf{ else } f\,(x * y)\,(y-1) \textbf{ in } f\ 1$$

$\text{code}_V\ e\ \varnothing\ 0$ emiteerib CBV korral koodi:

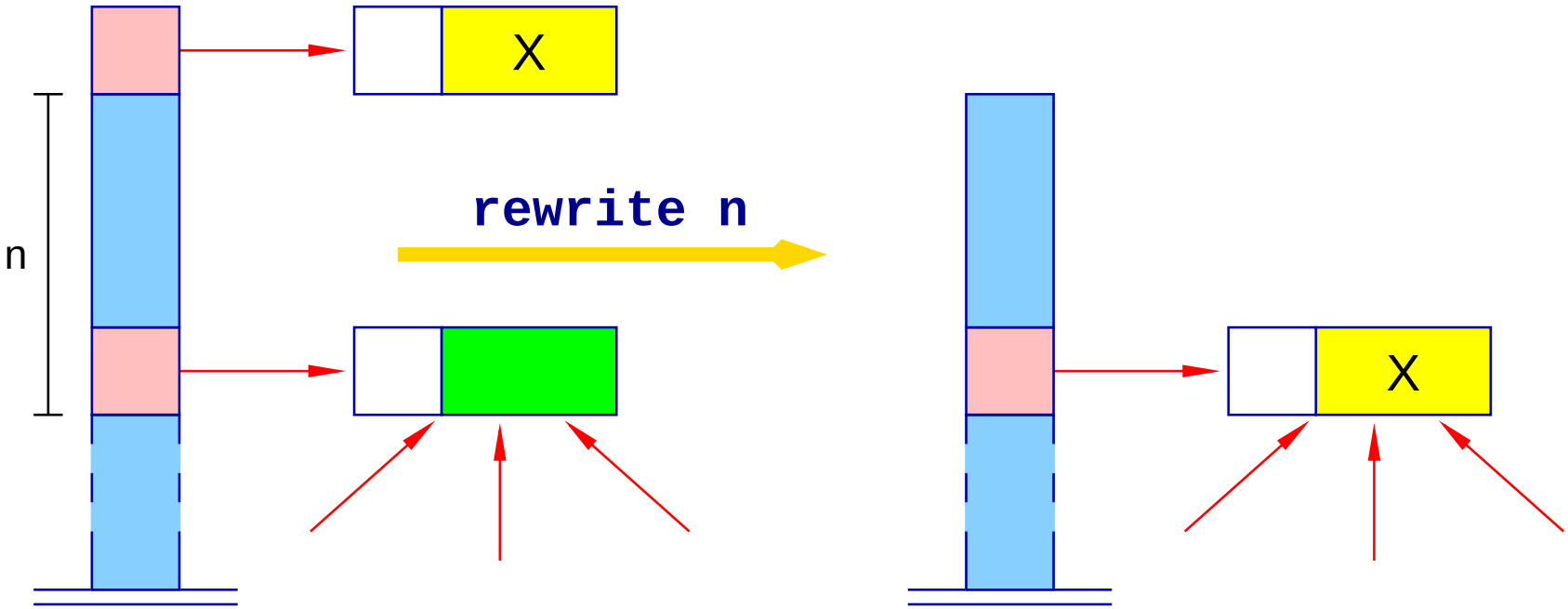
```
0  alloc 1          0  A: targ 2          4     loadc 1
1  pushloc 0        0     ...              5     mkbasic
2  mkvec 1          0     return 2         5     pushloc 4
2  mkfunval A       2  B: rewrite 1       6     apply
2  jump B           1     mark C           2  C: slide 1
```

Lokaalsete definitsioonide transleerimine




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

Lokaalsete definitsioonide transleerimine




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

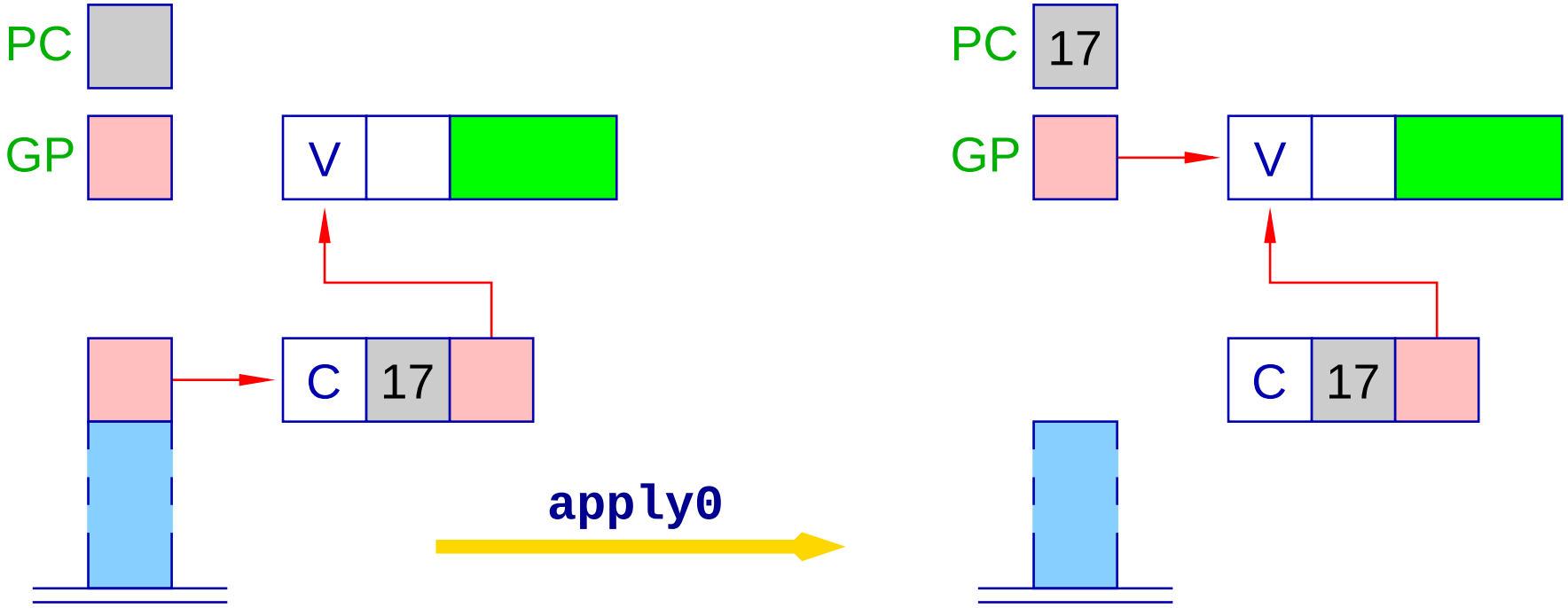
- Viit `S[SP-n]` jääb samaks!
- Ainult tema sisu muutub!

Sulundid ja nende väärtustamine

- Sulundeid on tarvis CBN semantika korral.
- Enne muutuja poole pöördumist peab tema väärtus olema saadaval.
- Vastasel korral tuleb temaga seotud sulund väärtustada.
- Sulund on sisuliselt parameetriteta funktsioon.
- Seega, väärtustamine on tema rakendamine 0 argumendile.
- Sulundi väärtustamine toimub käsu `eval` abil.

```
eval  =  if (S[SP]→tag = C) {
             mark PC;
             pushloc 3;
             apply0;
         }
```

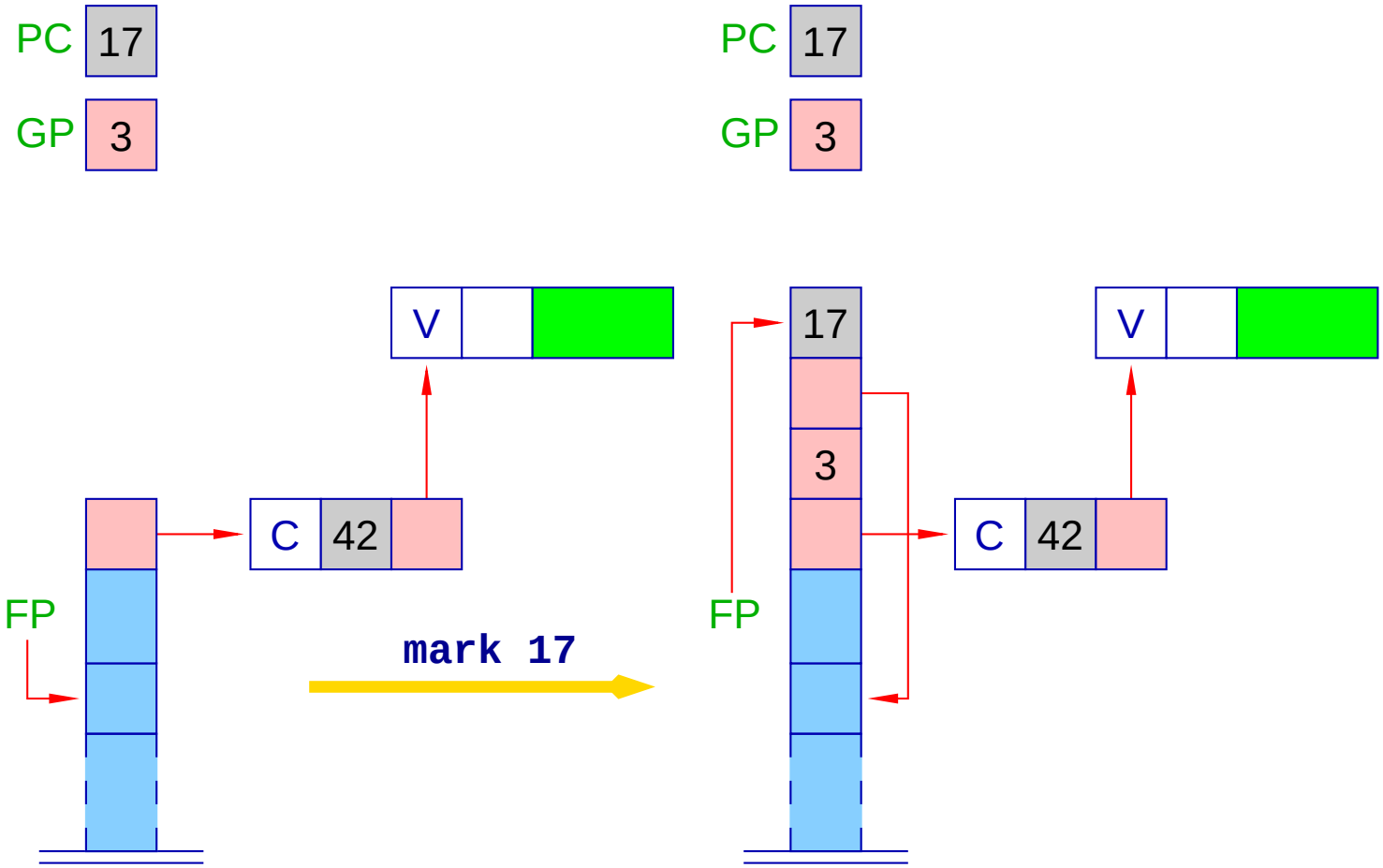
Sulundid ja nende väärtustamine




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

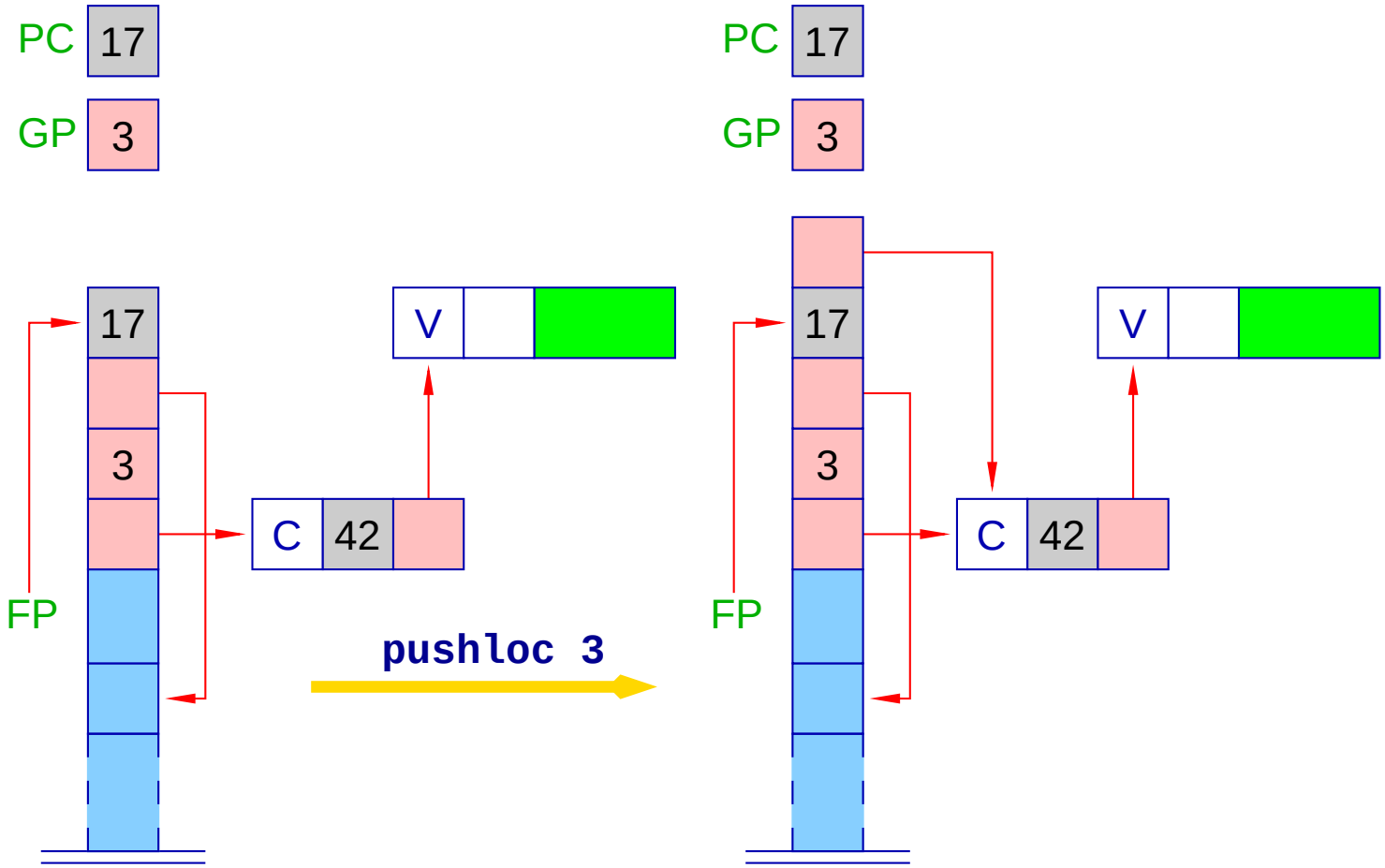
Sulundid ja nende väärtustamine

Sulundi väärtustamine käsu `eval` abil:



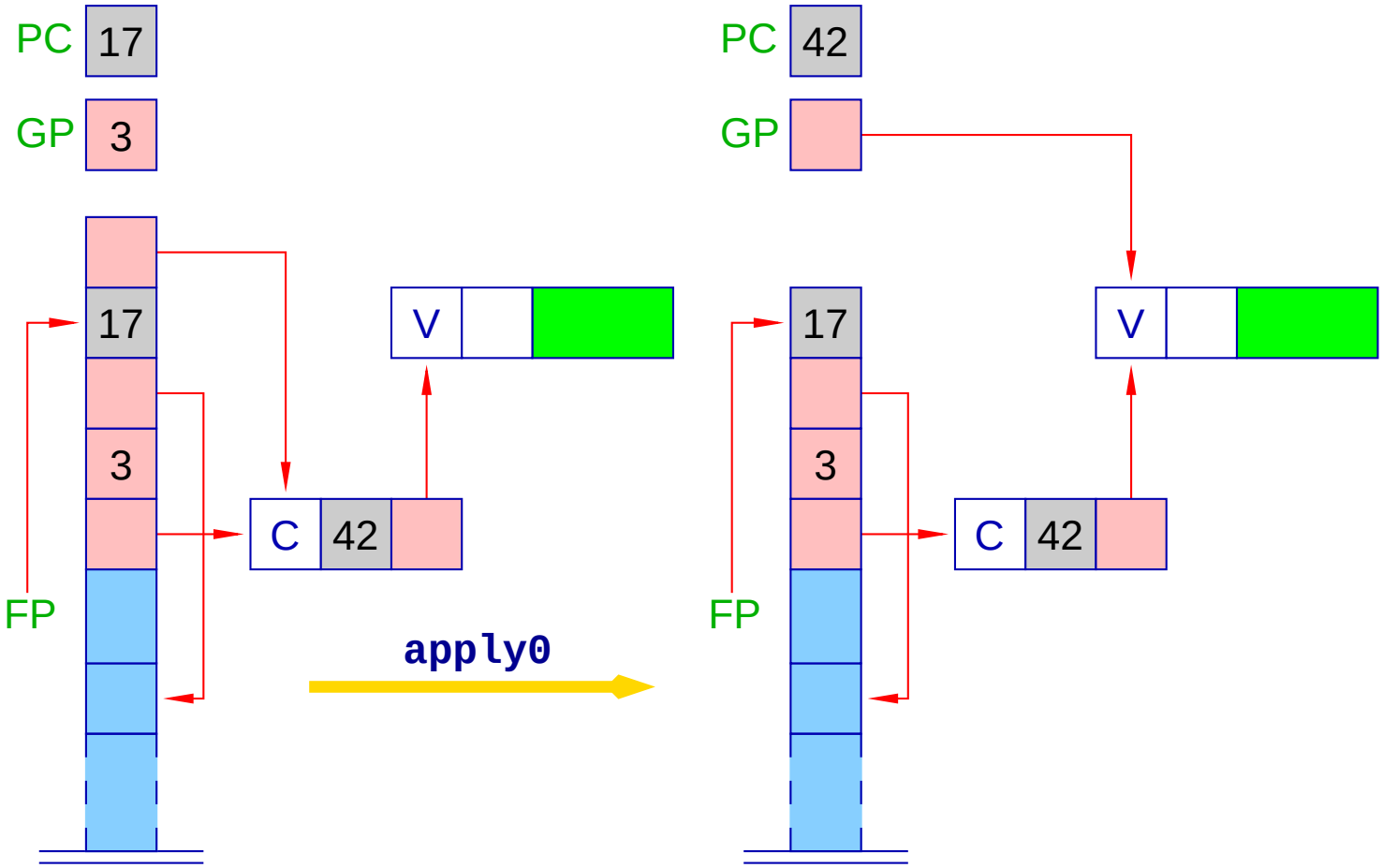
Sulundid ja nende väärtustamine

Sulundi väärtustamine käsu `eval` abil:



Sulundid ja nende väärtustamine

Sulundi väärtustamine käsu `eval` abil:



Sulundid ja nende väärtustamine

Avaldise e sulundi konstrueerimiseks:

- pakitakse temas esinevad vabad muutujad globaalvektorisse;
- luuakse C-objekt, mis viitab sellele vektorile ja avaldise väärtustamisele vastava koodi algusse.

$\text{code}_C\ e\ \rho\ \text{sd}$ =

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

kus $\{z_0, \ldots, z_{g-1}\} = \mathit{free}(e)$

$\rho' = \{z_i \mapsto (G, i) \mid i = 0, \ldots, g-1\}$

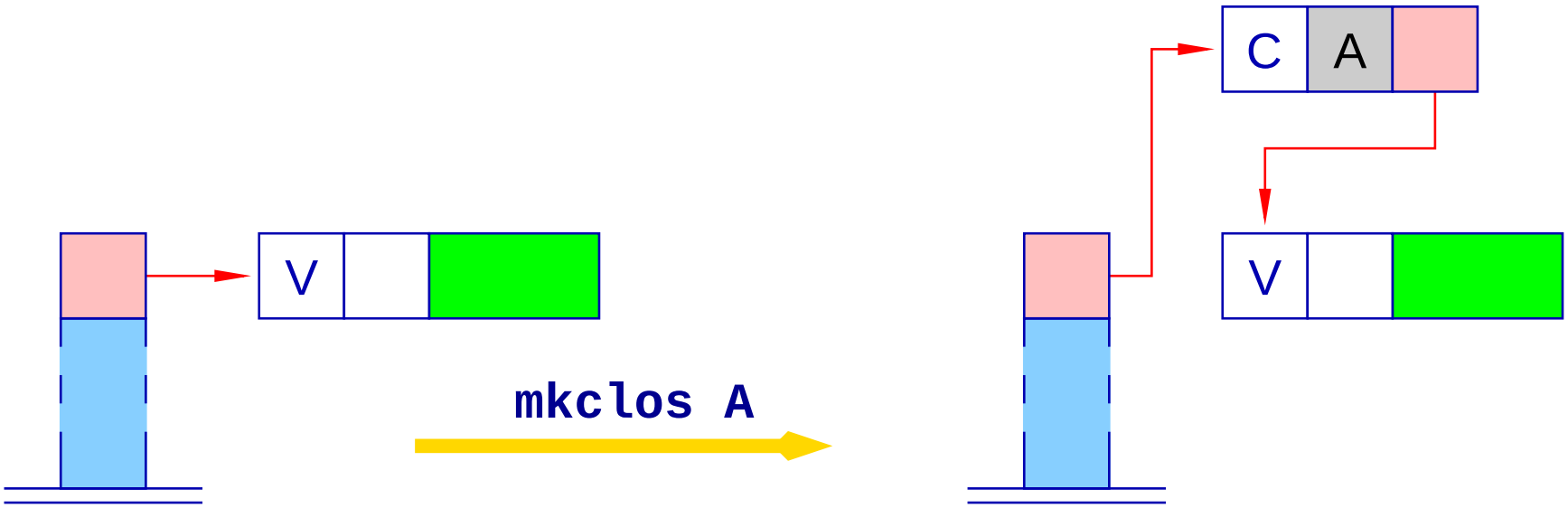
Sulundid ja nende väärtustamine

Näide: olgu $e \equiv a * a$ keskkonnas $\rho = \{a \mapsto (L, 0)\}$ ja $\mathrm{sd} = 1$.

$\mathrm{code}_C\ e\ \rho\ \mathrm{sd}$ emiteerib koodi:

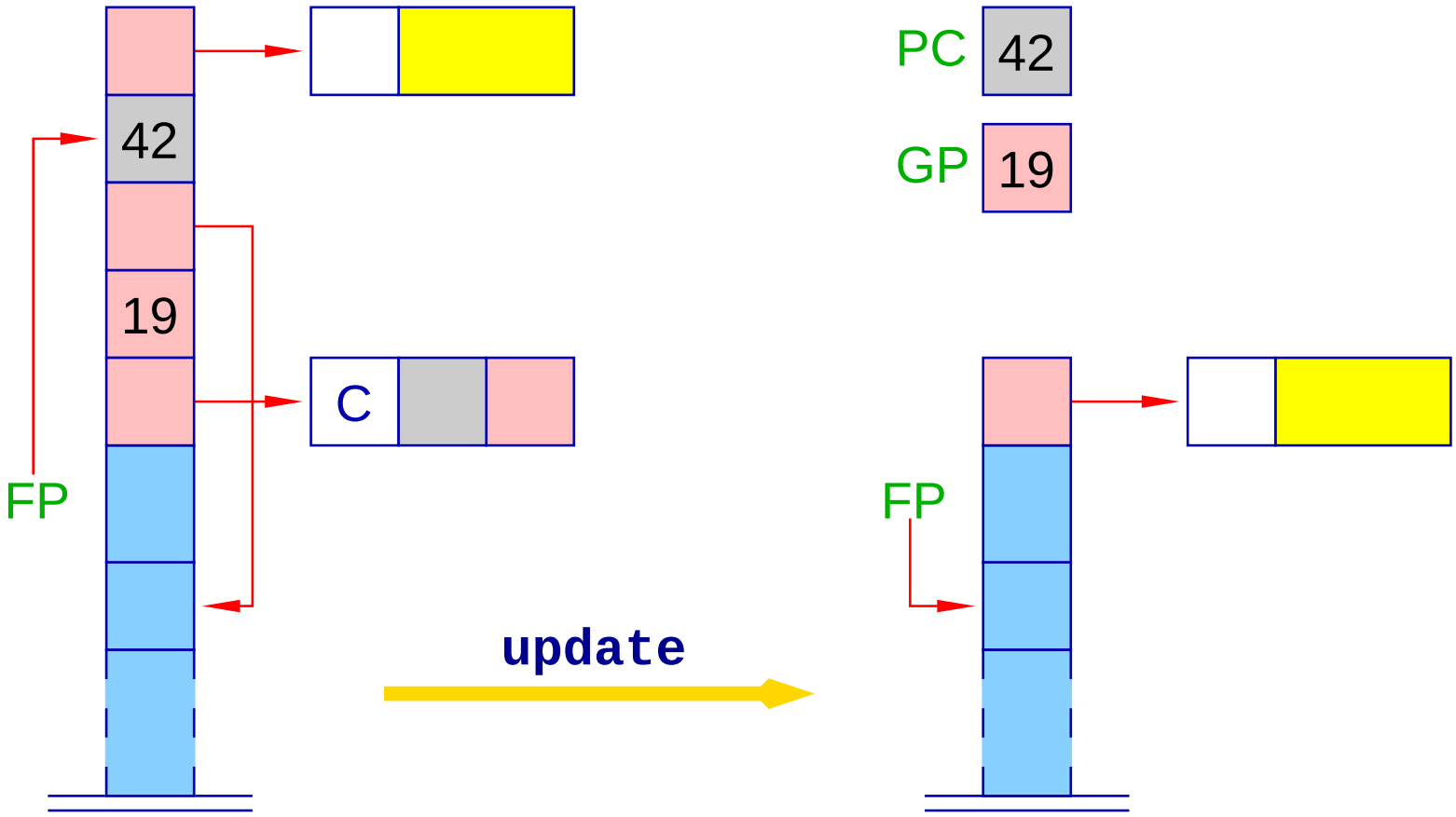
```
1   pushloc 1        0   A: pushglob 0      2       getbasic
2   mkvec 1          1      eval             2       mul
2   mkclos A         1      getbasic         1       mkbasic
2   jump B           2      pushglob 0       1       update
                     2      eval             2   B:  ...
```

Sulundid ja nende väärtustamine




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

Sulundid ja nende väärtustamine

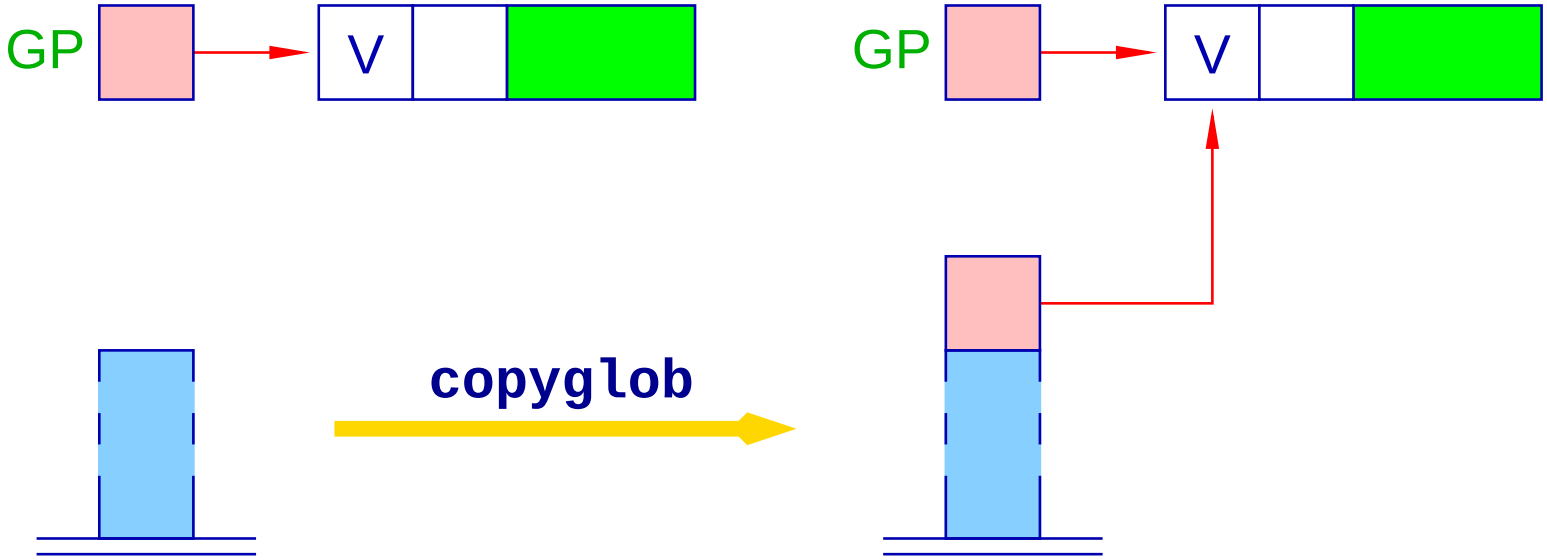


Optimeerimine I: Globaalsed muutujad

- Funktsionaalprogrammid konstrueerivad palju F- ja C-objekte.
- Muuhulgas tähendab see uute globaalvektorite loomist.
- "Globaalselt defineeritud" (top level) muutujatele võime staatiliselt määrata absoluutaadressid, mille kaudu neile ligi pääseb.
  - Kuna need absoluutaadressid on teada transleerimisajal, siis pole neid muutujaid vaja globaalvektoritesse lisada.
- Tihti on võimalik globaalvektoreid ka korduvkasutada.
  - Kasulik näiteks let-avaldiste või funktsiooniaplikatsioonide transleerimisel, kus võib konstrueerida ühe globaalvektori, mis sisaldab kõigi definitsioonide või argumentide vabu muutujaid.

Optimeerimine I: Globaalsed muutujad

Analoogselt lokaalsetele muutujatele, salvestatakse korduvkasutatavad globaalvektorid magasini.




```
SP++;
S[SP] = GP;
```

Optimeerimine I: Globaalsed muutujad

- Jagatavad globaalvektorid sisaldavad tihti rohkem muutujaid, kui on antud avaldises vabu muutujaid:
  - mida rohkem on vektoris muutujaid, seda suurema tõenäosusega on teda võimalik korduvkasutada.
- Liigsed muutujad globaalvektorites võivad takistada mälu vabastamist (space leaks).
- Võimalik lahendus: kustutada viit pärast tema "eluea" lõppu.

Optimeerimine II: Sulundid

- Avaldise e sulundi konstrueerimine lükkab antud avaldise väärtustamise edasi ajani kuni selle avaldise väärtust on tegelikult vaja.
- Kui avaldise väärtust pole tarvis, siis talle vastavat sulundit ka ei väärtustata (laisk väärtustamine).
- Kui aga staatiliselt on teada, et avaldise väärtust on kindlasti vaja (strictness analysis), siis on vahepealne sulundi konstrueerimine asjatu lisatöö.
- Seega, kui avaldis e on agaras kontekstis, siis:

$$\text{code}_C\ e\ \rho\ \text{sd} \quad = \quad \text{code}_V\ e\ \rho\ \text{sd}$$

- Sulundi konstrueerimine võib olla mõtetu ka juhul, kui avaldis on väga lihtne.

Optimeerimine II: Sulundid

Baasväärtused:

Sulundi konstrueerimine baasväärtusele on vähemalt sama kulukas, kui B-objekti enda konstrueerimine!
Seega:

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

See asendab koodijada:

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

Optimeerimine II: Sulundid

Muutujad:

Muutuja on seotud kas väärtuse või C-objektiga ning uue sulundi konstrueerimine on üleliigne. Seega:

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

See asendab koodijada:

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

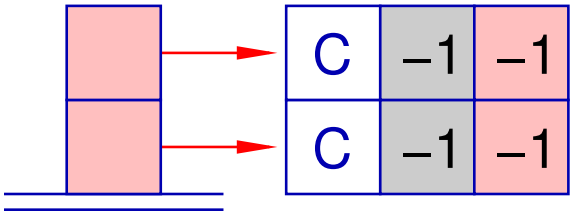
Näide: olgu $e \equiv$ **letrec** $a = b;\ b = 7$ **in** a, siis $\text{code}_V\ e\ \varnothing\ 0$ genereerib:

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

Optimeerimine II: Sulundid

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

alloc 2



Optimeerimine II: Sulundid

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

pushloc 0

Optimeerimine II: Sulundid

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

rewrite 2

Optimeerimine II: Sulundid

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

`loadc 7`

Optimeerimine II: Sulundid

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

mkbasic

Optimeerimine II: Sulundid

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

rewrite 1

Optimeerimine II: Sulundid

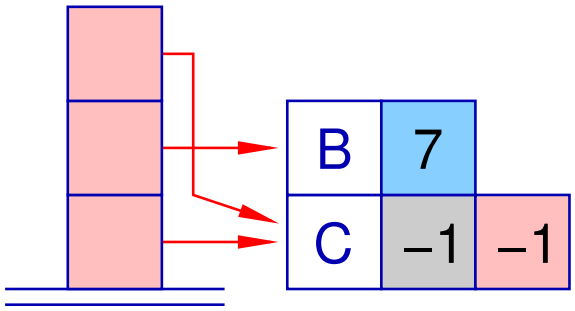
```
0  alloc 2        2  loadc 7       2  pushloc 1
2  pushloc 0      3  mkbasic       3  eval
3  rewrite 2      3  rewrite 1     3  slide 2
```

pushloc 1

Optimeerimine II: Sulundid

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

eval →



Segmentation Fault!!

Optimeerimine II: Sulundid

Ilmselt polnud optimeerimine päris korrektne!

Probleem:

Muutuja x seoti muutuja y väärtusega enne, kui see oli tegeliku väärtusega asendatud!!

Lahendus:

tsüklilised definitsioonid: mitte lubada definitsioone kujul

$$\textbf{letrec } a = b;\ \ldots;\ b = a \textbf{ in } \ldots$$

atsüklilised definitsioonid: järjestada definitsioonid ringi nii, et defineeritavatest muutujates sõltuvad definitsioonid oleksid tagapool.

Optimeerimine II: Sulundid

Funktsioonid:

Funktsioonid on väärtused ja neid edasi ei väärtustata. Selle asemel, et genereerida kood, mis konstrueerib F-objekti sulundi, võime otse konstrueerida F-objekti.

Seega:

$$\text{code}_C\ (\mathbf{fn}\ x_0, \ldots, x_{k-1} \Rightarrow e)\ \rho\ \text{sd} \quad = \quad \text{code}_V\ (\mathbf{fn}\ x_0, \ldots, x_{k-1} \Rightarrow e)\ \rho\ \text{sd}$$

Kogu programmi transleerimine

Abstraktse masina olek enne programmi käivitamist:

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

Programm (so. avaldis) e ei tohi sisaldada vabu muutujaid.

Genereeritakse kood, mis väärtustab avaldise e ja seejärel lõpetab masina töö käsuga `halt`:

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

Kogu programmi transleerimine

- Toodud transleerimisskeemid genereerivad "spagetikoodi".
- Põhjus: funktsioonikehade ja sulundite kood paigutatakse vahetult pärast `mkfunval` ja `mkclos` käske, hüpates üle selle koodi.
- Alternatiiv: panna see kood kuhugi mujale; näiteks pärast `halt` käsku:

  Eelis: saame lahti hüpetest pärast `mkfunval` ja `mkclos` käske.

  Puudus: transleerimisskeemid muutuvad keerulisemaks.
- Lahendus: eemaldame "spagetikoodi" koodi genereerimisele järgnevas optimeerimisfaasis.

Kogu programmi transleerimine

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

”Spagetikoodi” elimineerimisel saame:

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

Struktuursed andmetüübid

Laiendame funktsionaalset keelt PuF andmestruktuuridega:

ennikud:

$$\begin{array}{rcl} e & ::= & \ldots \mid (e_0, \ldots, e_{k-1}) \mid \#j\ e \\ & \mid & (\textbf{let}\ (x_0, \ldots, x_{k-1}) = e_1\ \textbf{in}\ e_0) \end{array}$$

listid:

$$\begin{array}{rcl} e & ::= & \ldots \mid [\,] \mid (e_1 : e_2) \\ & \mid & (\textbf{case}\ e_0\ \textbf{of}\ [\,] \to e_1;\ h : t \to e_2) \end{array}$$

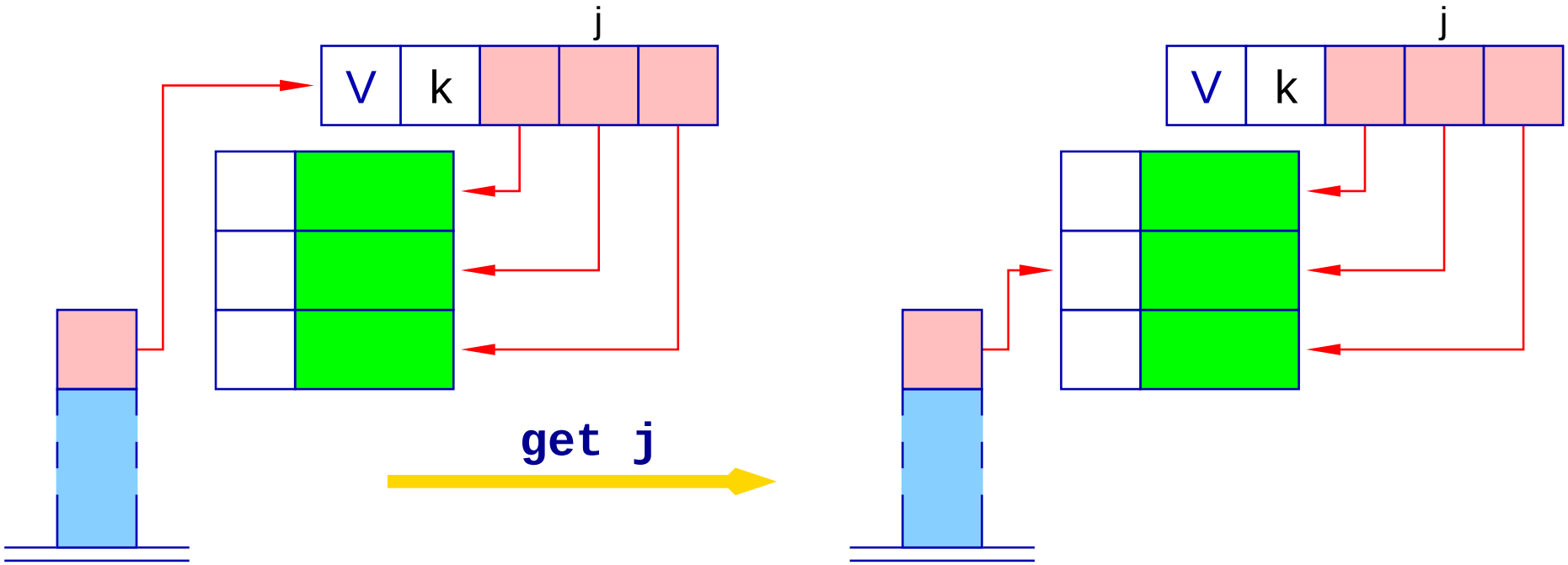
Struktuursed andmetüübid

Ennikute konstrueerimisel paigutatakse viidad komponentidele magasini ja seejärel luuakse vektor.
Konkreetse komponendi kättesaamiseks väärtustatakse ennik vektoriks ja seejärel väljastatakse vektori vastava indeksiga element.

$$
\begin{aligned}
\text{code}_V\ (e_0, \ldots, e_{k-1})\ \rho\ \text{sd} \quad = \quad & \text{code}_C\ e_0\ \rho\ \text{sd} \\
& \text{code}_C\ e_1\ \rho\ (\text{sd}+1) \\
& \ldots \\
& \text{code}_C\ e_{k-1}\ \rho\ (\text{sd}+k-1) \\
& \texttt{mkvec k} \\
\text{code}_V\ (\#j\ e)\ \rho\ \text{sd} \quad = \quad & \text{code}_V\ e\ \rho\ \text{sd} \\
& \texttt{get j}
\end{aligned}
$$

CBV korral väärtustatakse komponendid otse code_V abil.

Struktuursed andmetüübid




```
if (S[SP]→tag = V)
    S[SP] = S[SP]→v[j];
else Error ("Not Vector");
```

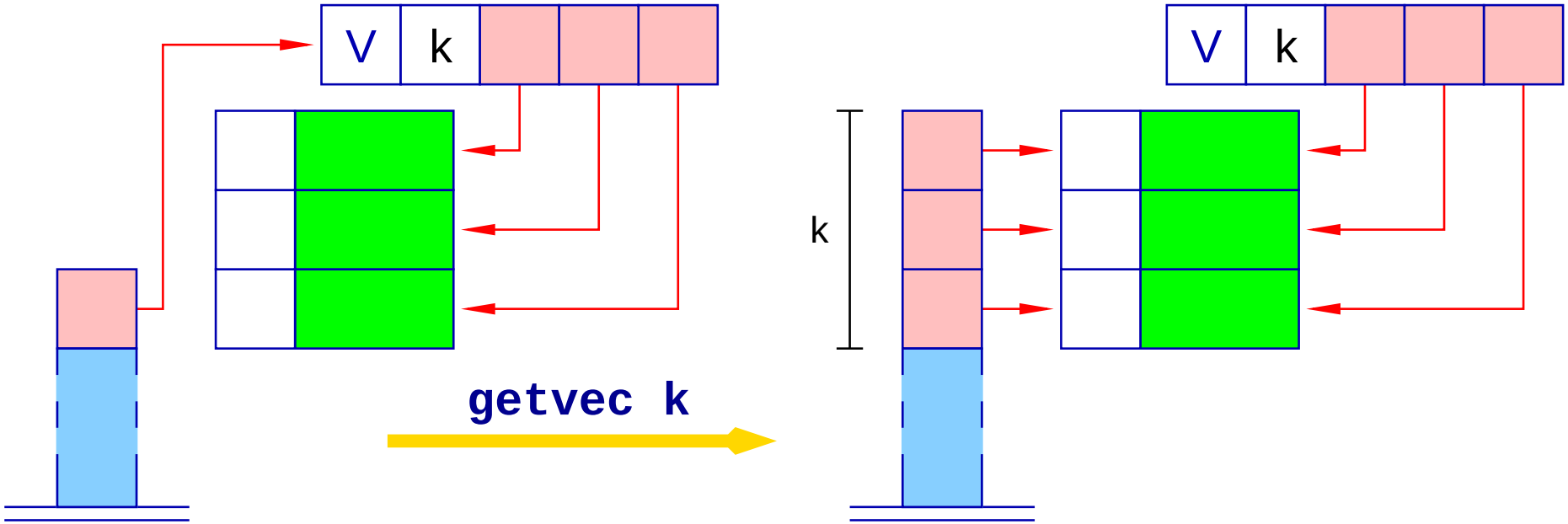
Struktuursed andmetüübid

Kõikide komponentide kättesaamiseks väärtustatakse ennik vektoriks ja seejärel lisatakse magasini viidad kõigile komponentidele.

$$\begin{aligned}\text{code}_V\ (\textbf{let}\ (y_0,\ldots,y_{k-1}) = e_1\ \textbf{in}\ e_0)\ \rho\ \text{sd} \quad = \quad & \text{code}_V\ e_1\ \rho\ \text{sd} \\ & \texttt{getvec k} \\ & \text{code}_V\ e_0\ \rho'\ \text{sd} \\ & \texttt{slide k}\end{aligned}$$

kus $\rho' = \rho \oplus \{y_i \mapsto \text{sd} + i \mid i = 0,\ldots,k-1\}$.

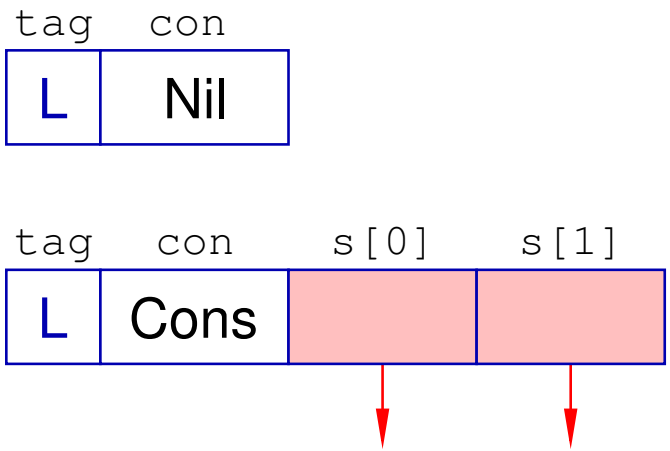
Struktuursed andmetüübid




```
if (S[SP]→tag = V)
    h = S[SP]; SP--;
    for (i=0; i≤k; i++) {
        SP++; S[SP] = h→v[i];
    }
else Error ("Not Vector");
```

Struktuursed andmetüübid

Listi konstruktorite esitamiseks kuhjas on uued objektid:



Empty List

Non–empty List

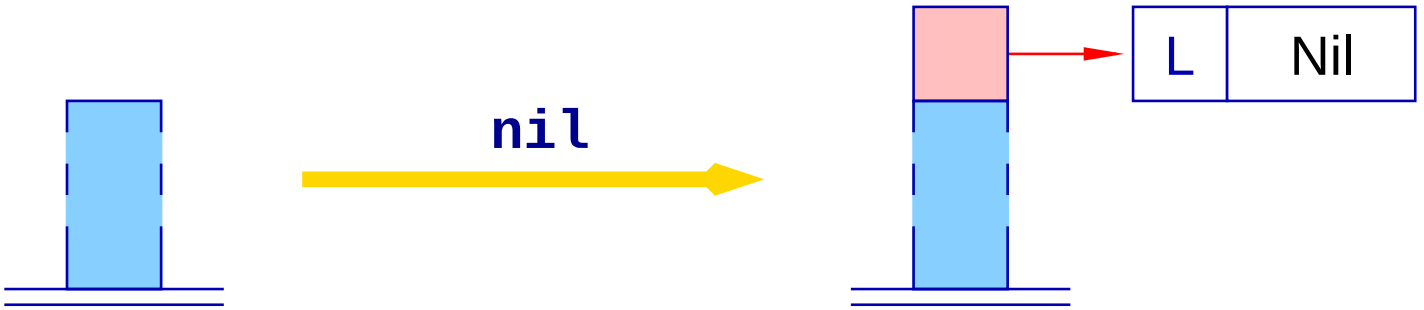
Struktuursed andmetüübid

Listi konstrueerimisel väärtustatakse argumendid (kui neid on; so. ":" korral) ning luuakse kuhjas konstruktorile vastav objekt:

$$
\begin{array}{lcl}
\text{code}_V\ [\,]\ \rho\ \text{sd} & = & \texttt{nil} \\
\text{code}_V\ (e_1 : e_2)\ \rho\ \text{sd} & = & \text{code}_C\ e_1\ \rho\ \text{sd} \\
 & & \text{code}_C\ e_2\ \rho\ (\text{sd}+1) \\
 & & \texttt{cons}
\end{array}
$$

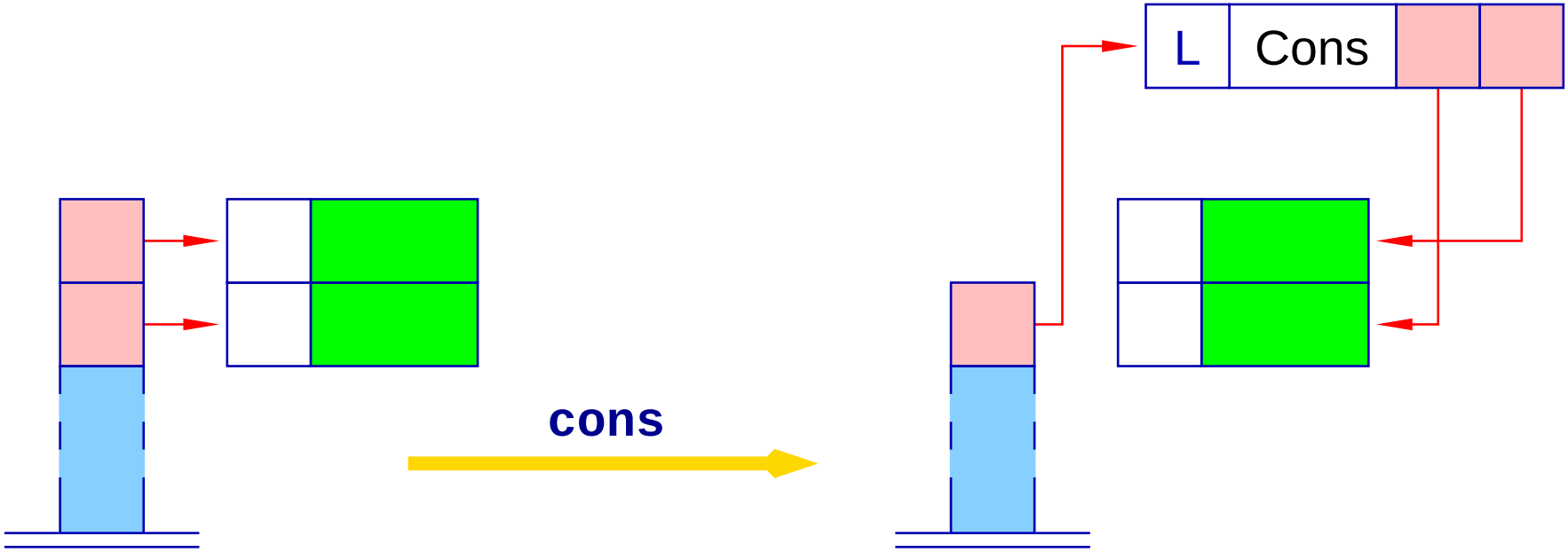
CBV korral väärtustatakse listi pea ja saba otse code_V abil.

Struktuursed andmetüübid




```
SP++;
S[SP] = new(L,Nil);
```

Struktuursed andmetüübid




```
S[SP-1] = new(L,Cons,S[SP-1],S[SP]);
SP--;
```

Struktuursed andmetüübid

- Listide inspekteerimine toimub näidiste sobitamise (pattern matching) abil.
- Case-avaldise $e \equiv \textbf{case}\ e_0\ \textbf{of}\ [\,] \to e_1;\ h : t \to e_2$ väärtustamiseks:
  - väärtustatakse avaldis e_0;
  - kui e_0 väärtus on tühi list, siis väärtustatakse avaldis e_1;
  - kui e_0 väärtus on mitte-tühi list, siis lisatakse listi peale ja sabale vastavate komponentide viidad magasini (so. seotakse muutujad h ja t) ning väärtustatakse avaldis e_2.

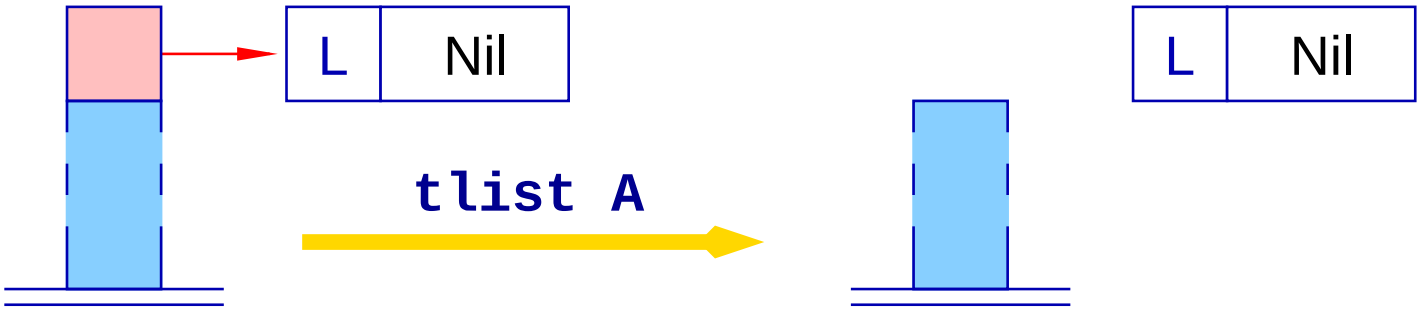
Struktuursed andmetüübid

$$\begin{aligned}
\text{code}_V\ (\textbf{case}\ e_0\ \textbf{of}\ [\,] \to e_1;\ h : t \to e_2)\ \rho\ \text{sd} \quad = \quad & \text{code}_V\ e_0\ \rho\ \text{sd} \\
& \texttt{tlist A} \\
& \text{code}_V\ e_1\ \rho\ \text{sd} \\
& \texttt{jump B} \\
\text{A:}\ & \text{code}_V\ e_2\ \rho'\ (\text{sd}+2) \\
& \texttt{slide 2} \\
\text{B:}\ & \ldots
\end{aligned}$$

kus $\rho' = \rho \oplus \{h \mapsto (L, \text{sd}+1), t \mapsto (L, \text{sd}+2)\}$.

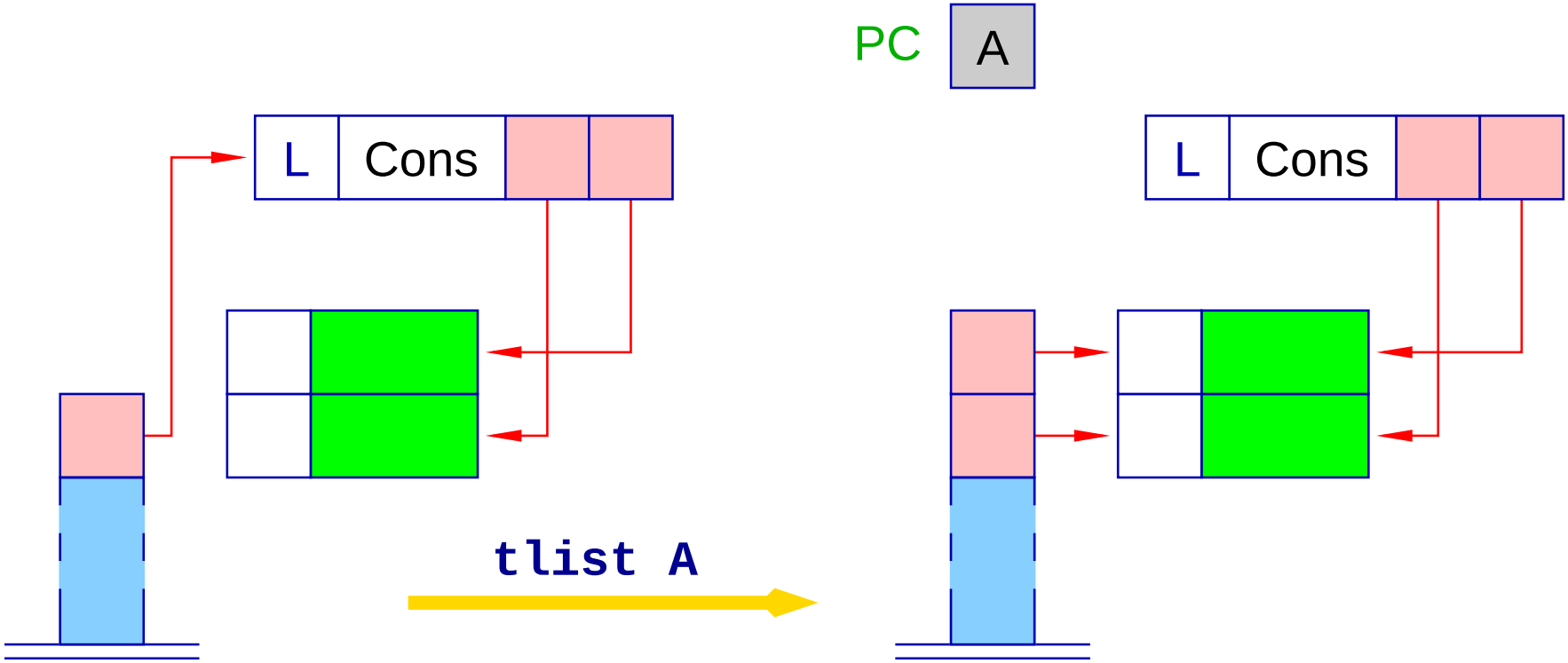
NB! On sama nii CBN kui CBV korral.

Struktuursed andmetüübid




```
if (S[SP]→tag ≠ L)
    Error ("Not List");
if (S[SP]→con = Nil)
    SP--;
```

Struktuursed andmetüübid




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

Struktuursed andmetüübid

Näide:

$$app = \textbf{fn}\ x, y \Rightarrow\ \textbf{case}\ x\ \textbf{of}$$
$$[\,] \quad \rightarrow \quad y$$
$$h : t \quad \rightarrow \quad h : (app\ t\ y)$$

```
0  targ 2
0  pushloc 0
1  eval
1  tlist A
0  pushloc 1
1  eval
1  jump B
```

```
2  A: pushloc 1
3     pushglob 0
4     pushloc 2
5     pushloc 6
6     mkvec 3
4     mkclos C
4     cons
0     slide 2
```

```
3  B: return 2
0  C: mark D
3     pushglob 2
4     pushglob 1
5     pushglob 0
6     eval
6     apply
1  D: update
```

Struktuursed andmetüübid

Sulundkontekstis enniku või listi korral võib sulundi konstrueerimise asemel luua vastavad objektid otse:

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

Sabarekursioon

- Funktsiooni aplikatsioon on mingis avaldises sabapositsioonis, kui tema väärtus võib olla kogu avaldise väärtuseks
  - aplikatsioon $r\ t\ (h : y)$ on sabapositsioonis avaldises:

$$\textbf{case}\ x\ \textbf{of}\ [\,] \to y;\ h : t \to r\ t\ (h : y)$$

  - aplikatsioon $f(x-1)$ ei ole sabapositsioonis avaldises:

$$\textbf{if}\ x \leq 1\ \textbf{then}\ 1\ \textbf{else}\ x * f(x-1)$$

- Funktsioon on sabarekursiivne, kui kõik tema (nii otse, kui kaudselt) rekursiivsed väljakutsed on sabapositsioonis.
- Sabapositsioonis oleva aplikatsiooni jaoks ei ole vaja uut freimi luua!

Sabarekursioon

Sabapositsioonis oleva aplikatsiooni $e'\ e_0\ \ldots\ e_{m-1}$ korral genereeritakse kood, mis:

- seob argumendid e_i formaalsete parameetritega ning väärtustab avaldise e' F-objektiks;
- vabastab aktiivse freimi lokaalsed muutujad;
- rakendab funktsiooni argumentidele.

NB! Argumentide ja funktsiooni väärtustamine toimub kehtivas aktiivses freimis.

Sabarekursioon

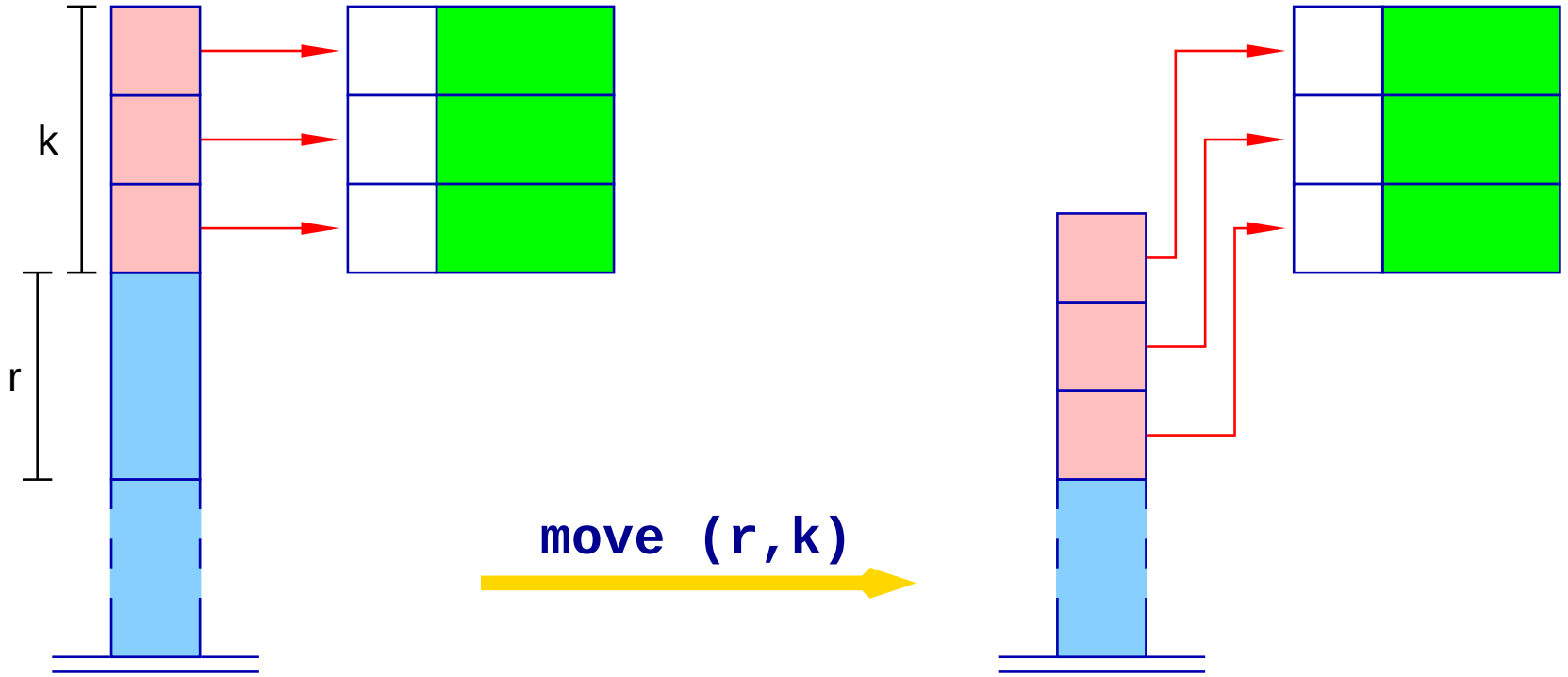
CBN korral genereeritakse kood:

$$
\begin{aligned}
\text{code}_V\ (e'\ e_0\ \ldots\ e_{m-1})\ \rho\ \text{sd} \quad = \quad & \text{code}_C\ e_{m-1}\ \rho\ \text{sd} \\
& \text{code}_C\ e_{m-2}\ \rho\ (\text{sd}+1) \\
& \ldots \\
& \text{code}_C\ e_0\ \rho\ (\text{sd}+m-1) \\
& \text{code}_V\ e'\ \rho\ (\text{sd}+m) \\
& \texttt{move}\ (\text{sd}+k,\ m+1) \\
& \texttt{apply}
\end{aligned}
$$

kus k on "välise" funktsiooni parameetrite arv.

CBV korral on argumentide e_i jaoks code_C asemel code_V.

Sabarekursioon




```
SP = SP - k - r;
for (i=1; i≤k; i++)
    S[SP+i] = S[SP+i+r];
SP = SP + k;
```

Sabarekursioon

Näide:
$$rev = \textbf{fn}\ x, y \Rightarrow\ \textbf{case}\ x\ \textbf{of}$$
$$[\,] \quad \rightarrow \quad y$$
$$h : t \quad \rightarrow \quad rev\ t\ (h : y)$$

Funktsiooni *rev* kehale vastab CBN korral kood:

```
0  targ 2          0     jump B          4     pushglob 0
0  pushloc 0                             5     eval
1  eval            2  A: pushloc 1       5     move (4,3)
1  tlist A         3     pushloc 4             apply
0  pushloc 1       4     cons
1  eval            3     pushloc 1       1  B: return 2
```

Kuna vanad organisatoorsed pesad on alles, siis on `return 2` saavutatav ainult otsehüppega tühjale listile vastavast harust.