31 Clauses

Clausal code must

- allocate stack space for locals;
- evaluate the body;
- free the stack frame (whenever possible :-)

Let r denote the clause: $p(X_1, \ldots, X_k) \leftarrow g_1, \ldots, g_n$.

Let $\{X_1, \ldots, X_m\}$ denote the set of locals of r and ρ the address environment:

$$\rho\ X_i = i$$

Remark: The first k locals are always the formals :-)

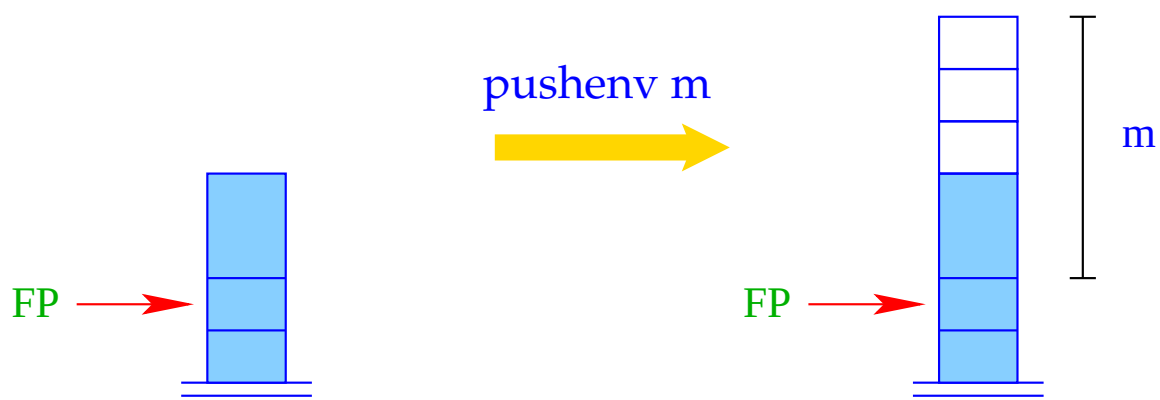
Then we translate:

$$
\begin{array}{lcll}
\text{code}_C\ r & = & \text{pushenv m} & \text{// allocates space for locals} \\
 & & \text{code}_G\ g_1\ \rho & \\
 & & \ldots & \\
 & & \text{code}_G\ g_n\ \rho & \\
 & & \text{popenv} &
\end{array}
$$

The instruction popenv restores FP and PC and tries to pop the current stack frame.

It should succeed whenever program execution will never return to this stack frame :-)

The instruction pushenv m sets the stack pointer:



SP = FP + m;

Example:

Consider the clause r:

$$\mathsf{a}(X, Y) \leftarrow \mathsf{f}(\bar{X}, X_1), \mathsf{a}(\bar{X}_1, \bar{Y})$$

Then $\text{code}_C\ r$ yields:

```
pushenv 3      mark A      A:  mark B      B:  popenv
               putref 1        putref 3
               putvar 3        putref 2
               call f/2        call a/2
```

32 Predicates

A predicate q/k is defined through a sequence of clauses $\quad rr \equiv r_1 \ldots r_f$.

The translation of q/k provides the translations of the individual clauses r_i.

In particular, we have for $\quad f = 1 \quad$:

$$\text{code}_P \; rr \quad = \quad \text{code}_C \; r_1$$

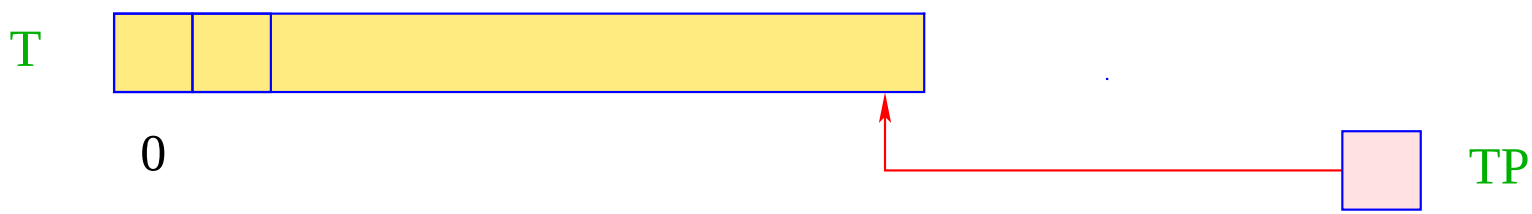
If q/k is defined through several clauses, the first alternative must be tried.

On failure, the next alternative must be tried

$\Longrightarrow$ backtracking :-)

32.1 Backtracking

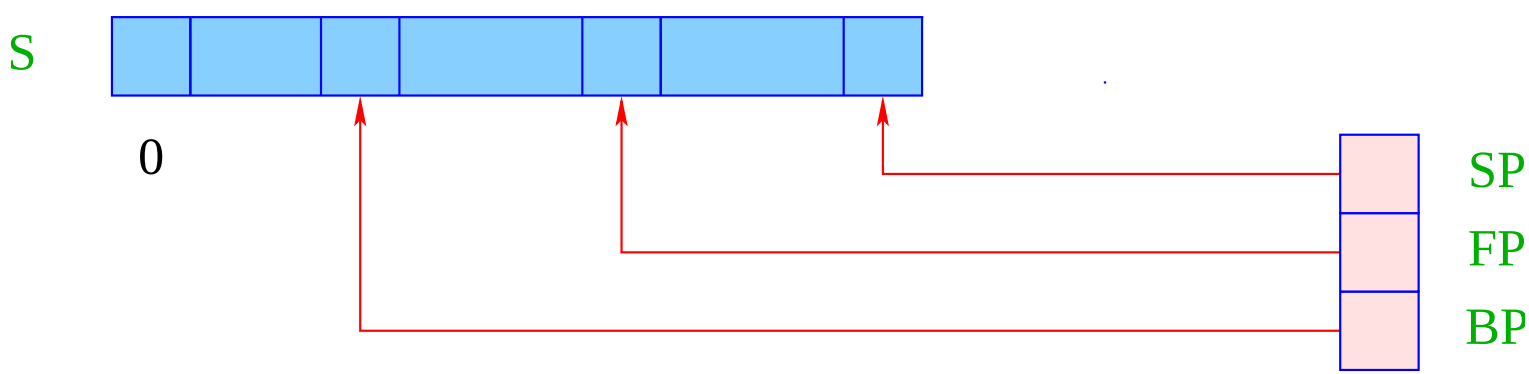
- Whenever unifcation fails, we call the run-time function `backtrack()`.
- The goal is to roll back the whole computation to the (dynamically :-) latest goal where another clause can be chosen $\Longrightarrow$ the last backtrack point.
- In order to undo intermediate variable bindings, we always have recorded new bindings with the run-time function `trail()`.
- The run-time function `trail()` stores variables in the data-structure trail:



TP ═══ Trail Pointer

points to the topmost occupied Trail cell

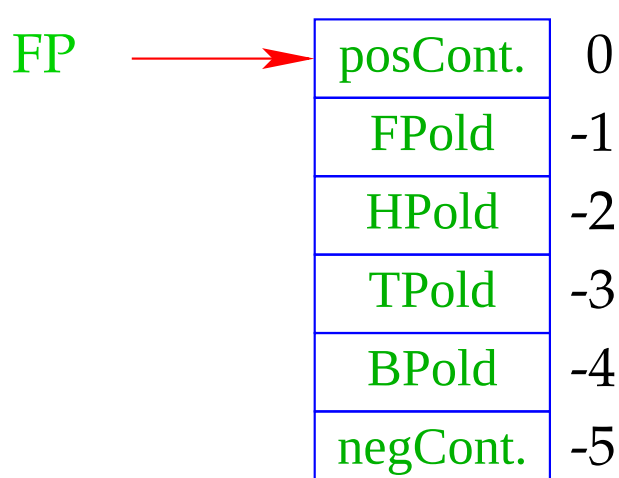
The current stack frame where backtracking should return to is pointed at by the extra register BP:



A backtrack point is stack frame to which program execution possibly returns.

- We need the code address for trying the next alternative (negative continuation address);
- We save the old values of the registers HP, TP and BP.
- Note: The new BP will receive the value of the current FP :-)

For this purpose, we use the corresponding four organizational cells:



For more comprehensible notation, we thus introduce the macros:

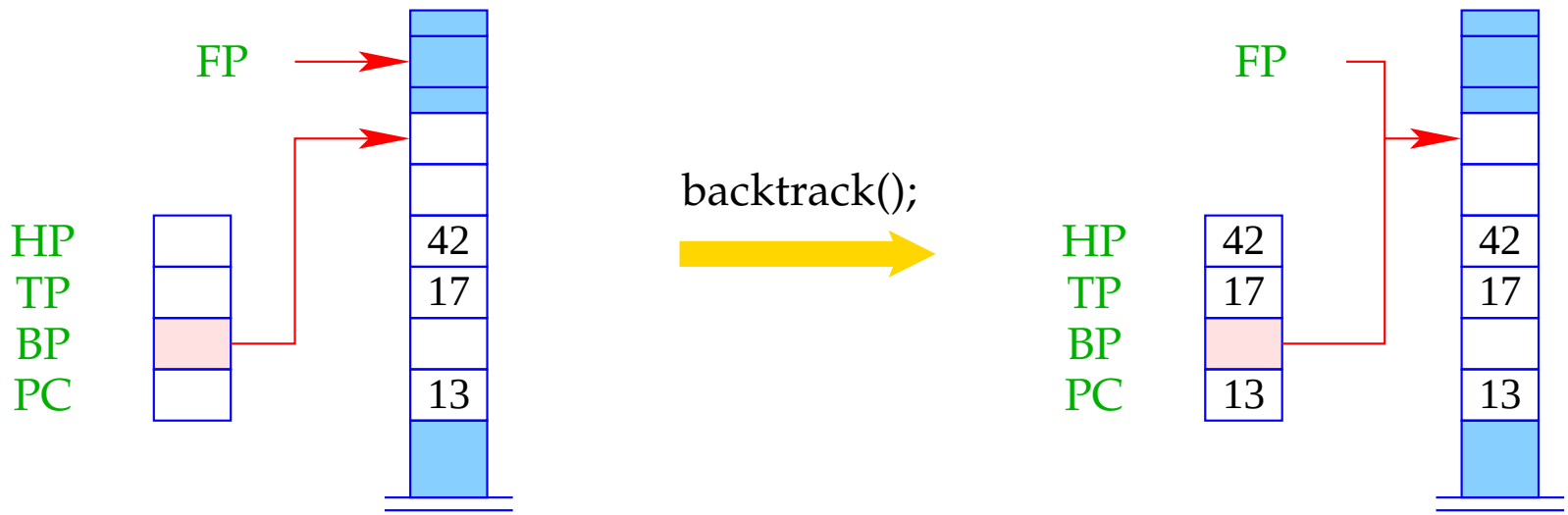
$$
\begin{aligned}
\text{posCont} &\equiv S[\text{FP}] \\
\text{FPold} &\equiv S[\text{FP}-1] \\
\text{HPold} &\equiv S[\text{FP}-2] \\
\text{TPold} &\equiv S[\text{FP}-3] \\
\text{BPold} &\equiv S[\text{FP}-4] \\
\text{negCont} &\equiv S[\text{FP}-5]
\end{aligned}
$$

for the corresponding addresses.

Remark:

$$
\begin{aligned}
\text{Occurrence on the left} &\Longrightarrow \text{saving the register} \\
\text{Occurrence on the right} &\Longrightarrow \text{restoring the register}
\end{aligned}
$$

Calling the run-time function `void backtrack()` yields:




```
void backtrack() {
    FP = BP; HP = HPold;
    reset (TPold, TP);
    TP = TPold; PC = negCont;
}
```

where the run-time function `reset()` undoes the bindings of variables established since the backtrack point.

32.2 Resetting Variables

Idea:

- The variables which have been created since the last backtrack point can be removed together with their bindings by popping the heap !!! :-)
- This works fine if younger variables always point to older objects.
- Bindings of old variables to younger objects, though, must be reset manually :-(
- These are therefore recorded in the trail.

Functions `void trail(ref u)` and `void reset (ref y, ref x)` can thus be implemented as:

```
void trail (ref u) {
   if (u < S[BP-2]) {
      TP = TP+1;
      T[TP] = u;
   }
}
```

```
void reset (ref x, ref y) {
   for (ref u=y; x<u; u--)
      H[T[u]] = (R,T[u]);
}
```

Here, `S[BP-2]` represents the heap pointer when creating the last backtrack point.

32.3 Wrapping it Up

Assume that the predicate q/k is defined by the clauses $r_1, \ldots, r_f$ $(f > 1)$.
We provide code for:

- setting up the backtrack point;
- successively trying the alternatives;
- deleting the backtrack point.

This means:

$$
\begin{array}{rcll}
\text{code}_P\ rr & = & \text{q/k}: & \text{setbtp} \\
 & & & \text{try } A_1 \\
 & & & \ldots \\
 & & & \text{try } A_{f-1} \\
 & & & \text{delbtp} \\
 & & & \text{jump } A_f \\
 & & A_1: & \text{code}_C\ r_1 \\
 & & & \ldots \\
 & & A_f: & \text{code}_C\ r_f
\end{array}
$$

Note:

- We delete the backtrack point before the last alternative :-)
- We jump to the last alternative — never to return to the present frame :-))

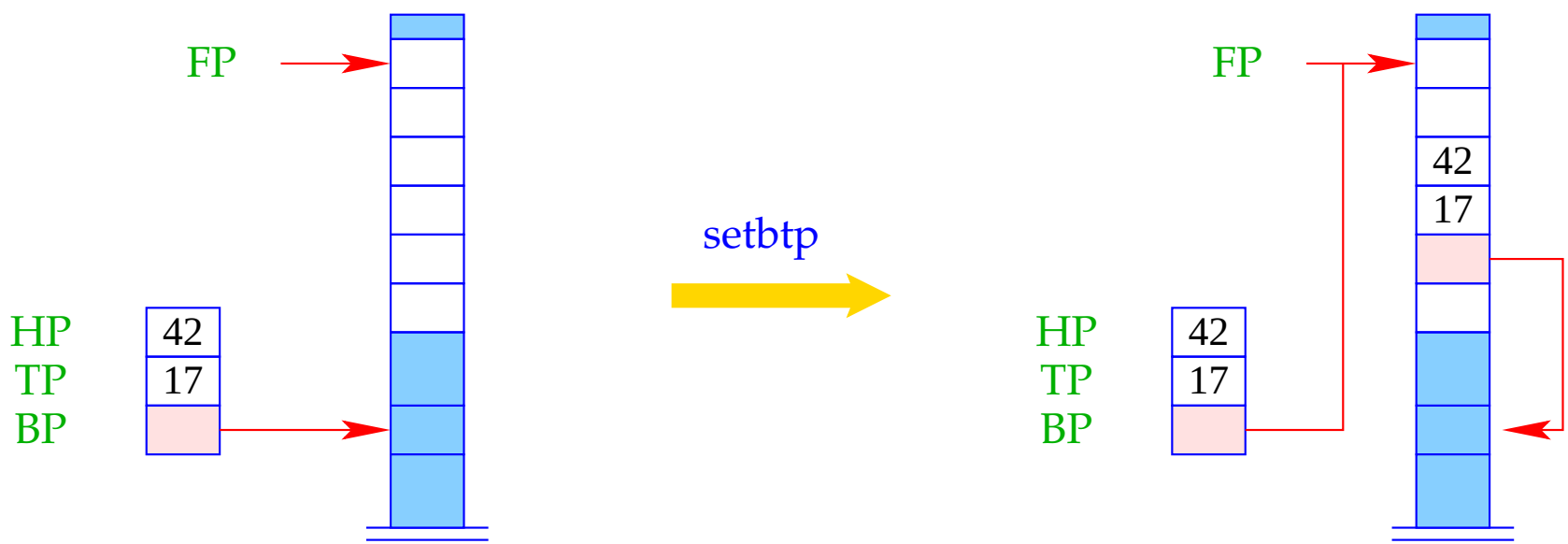
Example:

$$\begin{array}{lcl} \mathsf{s}(X) & \leftarrow & \mathsf{t}(\bar{X}) \\ \mathsf{s}(X) & \leftarrow & \bar{X} = a \end{array}$$

The translation of the predicate s yields:

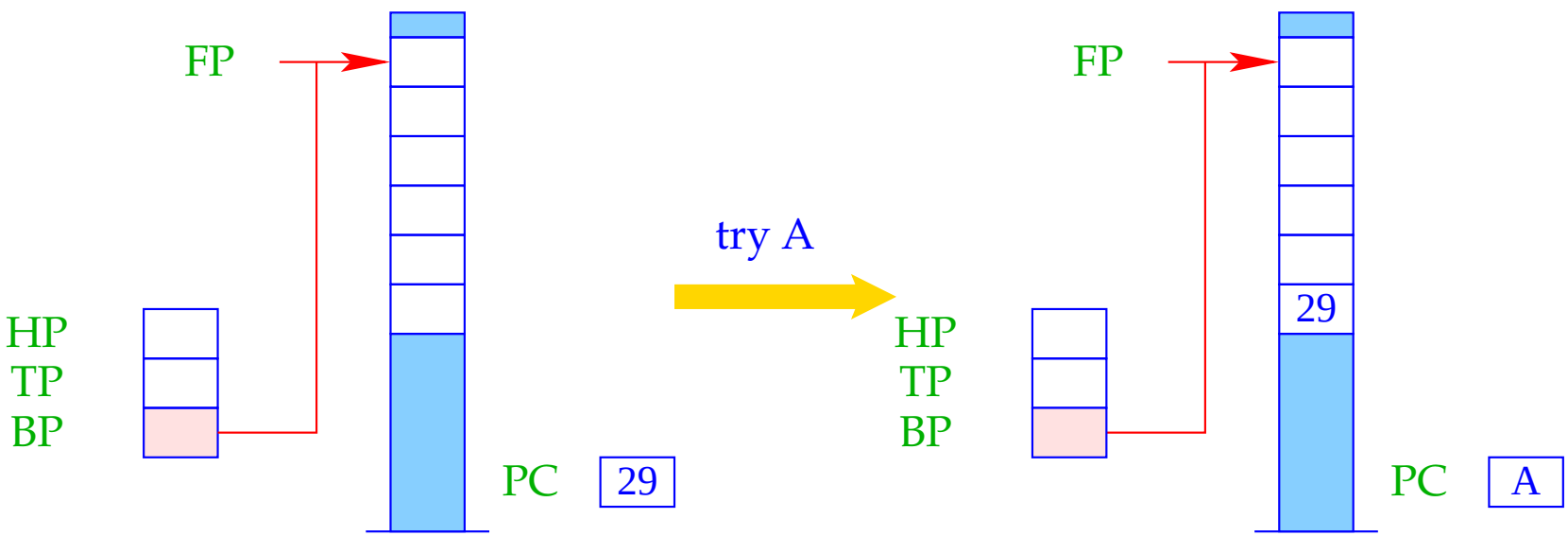
```
s/1:  setbtp    A:  pushenv 1   B:  pushenv 1
      try A         mark C          putref 1
      delbtp        putref 1        uatom a
      jump B        call t/1        popenv
                C:  popenv
```

The instruction setbtp saves the registers HP, TP, BP:



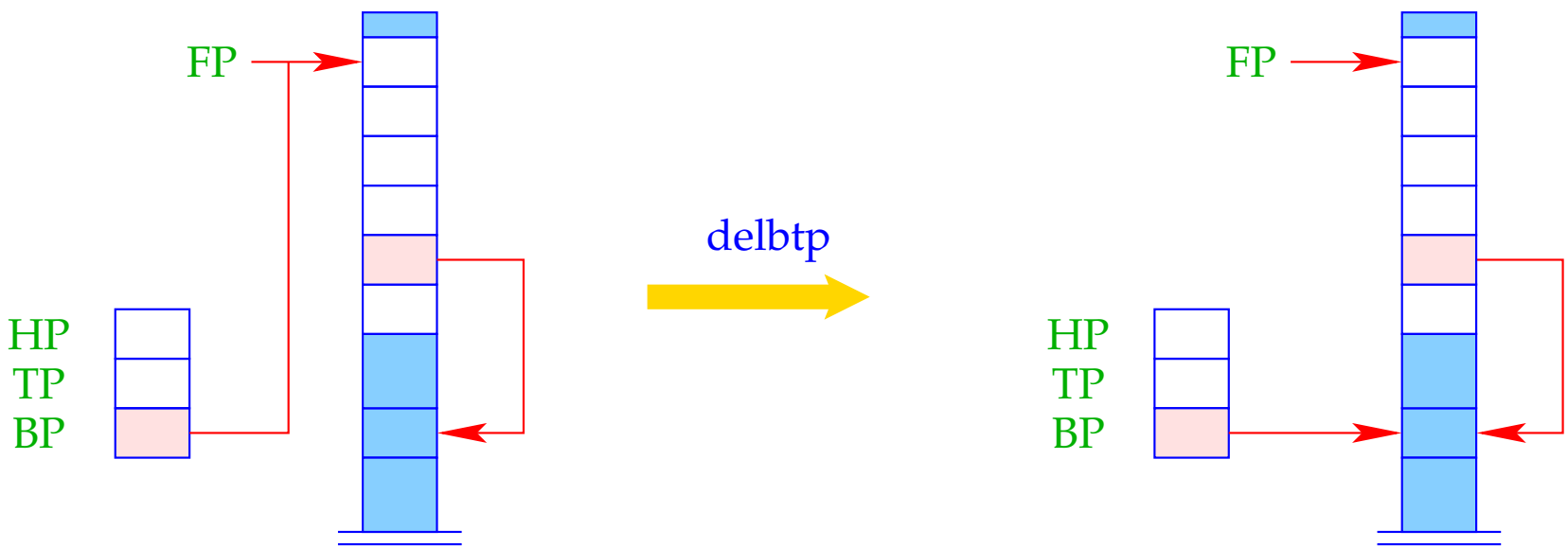

```
HPold = HP;
TPold = TP;
BPold = BP;
BP = FP;
```

The instruction try A tries the alternative at address A and updates the negative continuation address to the current PC:



negForts = PC;
PC = A;

The instruction delbtp restores the old backtrack pointer:



BP = BPold;

32.4 Popping of Stack Frames

Recall the translation scheme for clauses:

$$
\begin{aligned}
\text{code}_C\ r \quad = \quad & \text{pushenv m} \\
& \text{code}_G\ g_1\ \rho \\
& \ldots \\
& \text{code}_G\ g_n\ \rho \\
& \text{popenv}
\end{aligned}
$$

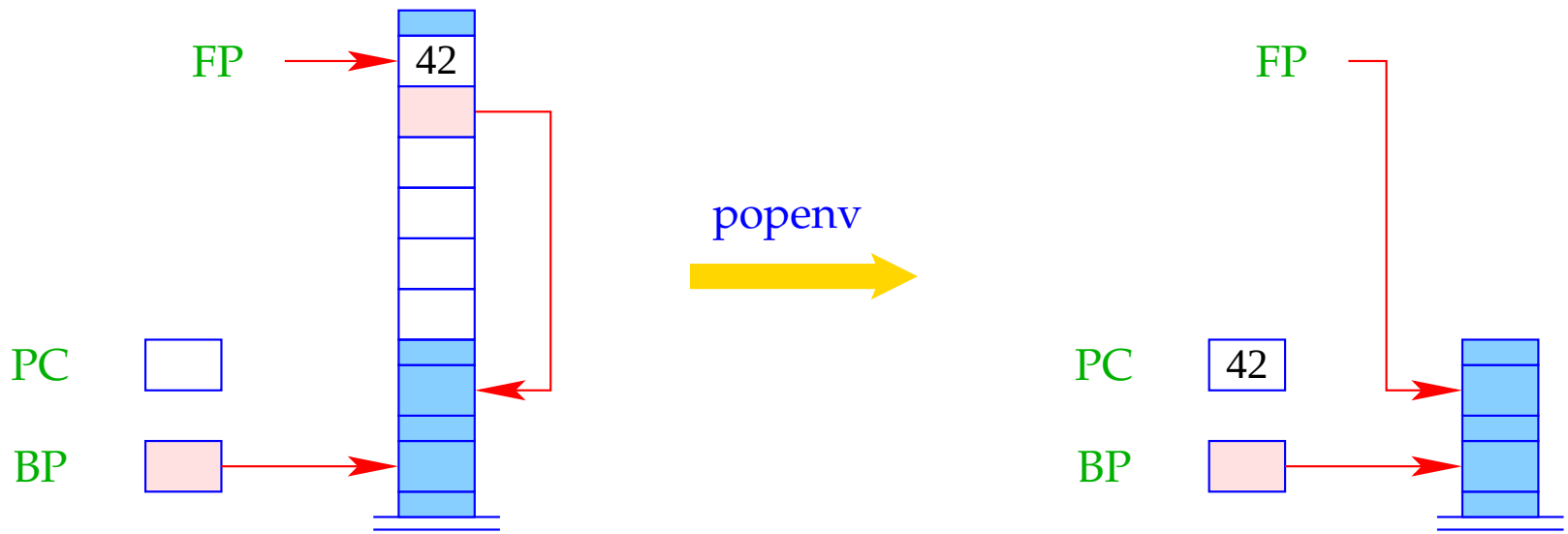
The present stack frame can be popped ...

- if the applied clause was the last (or only); and
- if all goals in the body are definitely finished.

$\Longrightarrow$ the backtrack point is older :-)

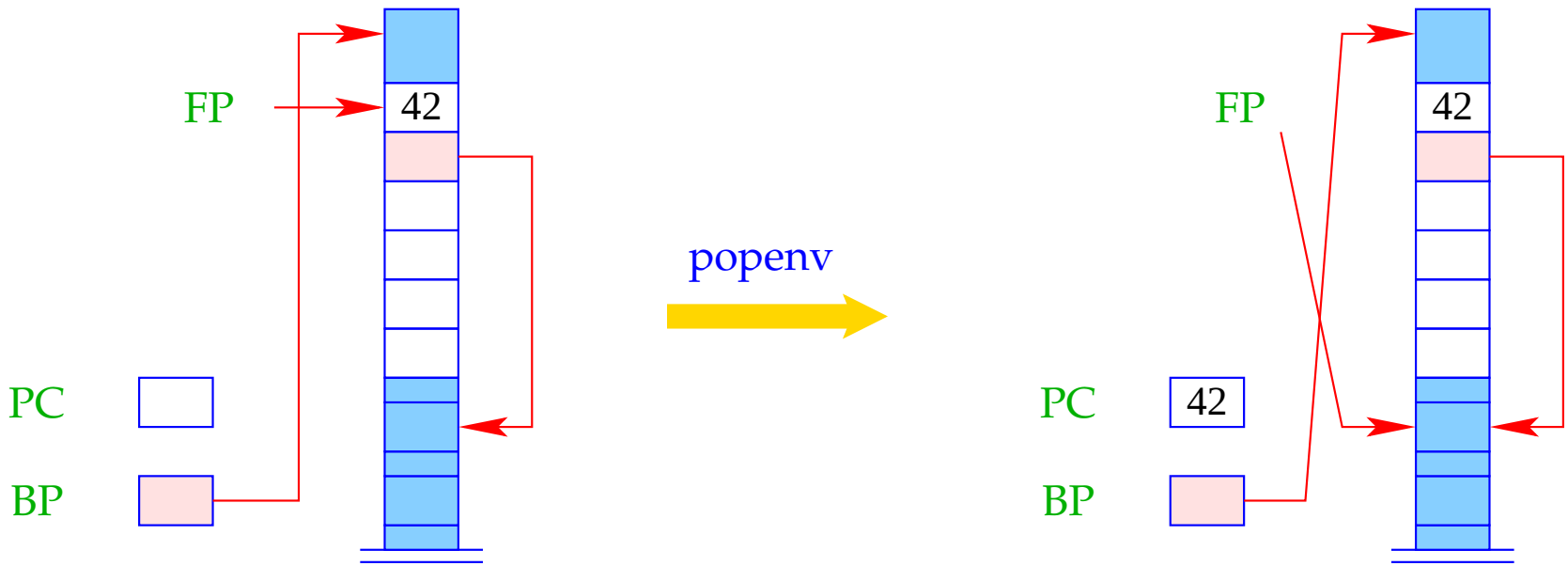
$\Longrightarrow$ FP > BP

The instruction popenv restores the registers FP and PC and possibly pops the stack frame:




```
if (FP > BP) SP = FP - 6;
PC = posCont;
FP = FPold;
```

Warning: popenv may fail to de-allocate the frame !!!



if (FP > BP) SP = FP - 6;
PC = posCont;
FP = FPold;

If popping the stack frame fails, new data are allocated on top of the stack. When returning to the frame, the locals still can be accessed through the FP :-))

33 Queries and Programs

The translation of a program: $p \equiv rr_1 \ldots rr_h ? g$
consists of:

- an instruction no for failure;
- code for evaluating the query g;
- code for the predicate definitions rr_i.

Preceding query evaluation:

$\Longrightarrow$ initialization of registers
$\Longrightarrow$ allocation of space for the globals

Succeeding query evaluation:

$\Longrightarrow$ returning the values of globals

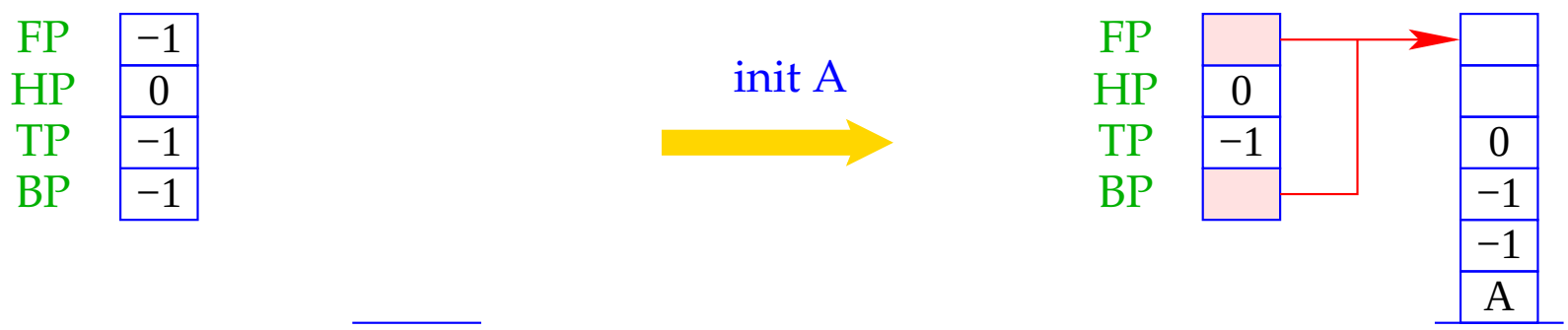
$$
\begin{array}{rcl}
\text{code } p & = & \text{init A} \\
 & & \text{pushenv d} \\
 & & \text{code}_G\ g\ \rho \\
 & & \text{halt d} \\
 & & \text{A: no} \\
 & & \text{code}_P\ rr_1 \\
 & & \ldots \\
 & & \text{code}_P\ rr_h
\end{array}
$$

where $free(g) = \{X_1, \ldots, X_d\}$ and ρ is given by $\rho\ X_i = i$.

The instruction halt d ...

- ... terminates the program execution;
- ... returns the bindings of the d globals;
- ... causes backtracking — if demanded by the user :-)

The instruction init A is defined by:



BP = FP = SP = 5;
S[0] = A;
S[1] = S[2] = -1;
S[3] = 0;
BP = FP;

At address "A" for a failing goal we have placed the instruction no for printing no to the standard output and halt :-)

The Final Example:

$$t(X) \leftarrow \bar{X} = b \qquad q(X) \leftarrow s(\bar{X}) \qquad s(X) \leftarrow \bar{X} = a$$

$$p \leftarrow q(X), t(\bar{X}) \qquad s(X) \leftarrow t(\bar{X}) \qquad ? \quad p$$

The translation yields:

```
      init N          popenv          q/1:  pushenv 1       E:  pushenv 1
      pushenv 0  p/0: pushenv 1             mark D              mark G
      mark A          mark B                putref 1            putref 1
      call p/0        putvar 1              call s/1            call t/1
A:    halt 0          call q/1        D:    popenv        G:    popenv
N:    no         B:   mark C          s/1:  setbtp        F:    pushenv 1
t/1:  pushenv 1       putref 1              try E               putref 1
      putref 1        call t/1              delbtp              uatom a
      uatom b    C:   popenv                jump F              popenv
```

34 Last Call Optimization

Consider the app predicate from the beginnning:

$$\begin{array}{lcl} \mathsf{app}(X, Y, Z) & \leftarrow & X = [\,],\ Y = Z \\ \mathsf{app}(X, Y, Z) & \leftarrow & X = [H|X'],\ Z = [H|Z'],\ \mathsf{app}(X', Y, Z') \end{array}$$

We observe:

- The recursive call occurs in the last goal of the clause.
- Such a goal is called last call.

$\Longrightarrow$ we try to evaluate it in the current stack frame !!!

$\Longrightarrow$ after (successful) completion, we will not return to the current caller !!!

Consider a clause r: $p(X_1, \ldots, X_k) \leftarrow g_1, \ldots, g_n$
with m locals where $g_n \equiv q(t_1, \ldots, t_h)$. The interplay between code_C and code_G:

$$
\begin{array}{rcl}
\text{code}_C\ r & = & \text{pushenv m} \\
 & & \text{code}_G\ g_1\ \rho \\
 & & \ldots \\
 & & \text{code}_G\ g_{n-1}\ \rho \\
 & & \text{mark B} \\
 & & \text{code}_A\ t_1\ \rho \\
 & & \ldots \\
 & & \text{code}_A\ t_h\ \rho \\
 & & \text{call q/h} \\
 & & \text{B :}\ \ \text{popenv}
\end{array}
$$

Replacement:	mark B	$\Longrightarrow$	lastmark
	call q/h; popenv	$\Longrightarrow$	lastcall q/h m

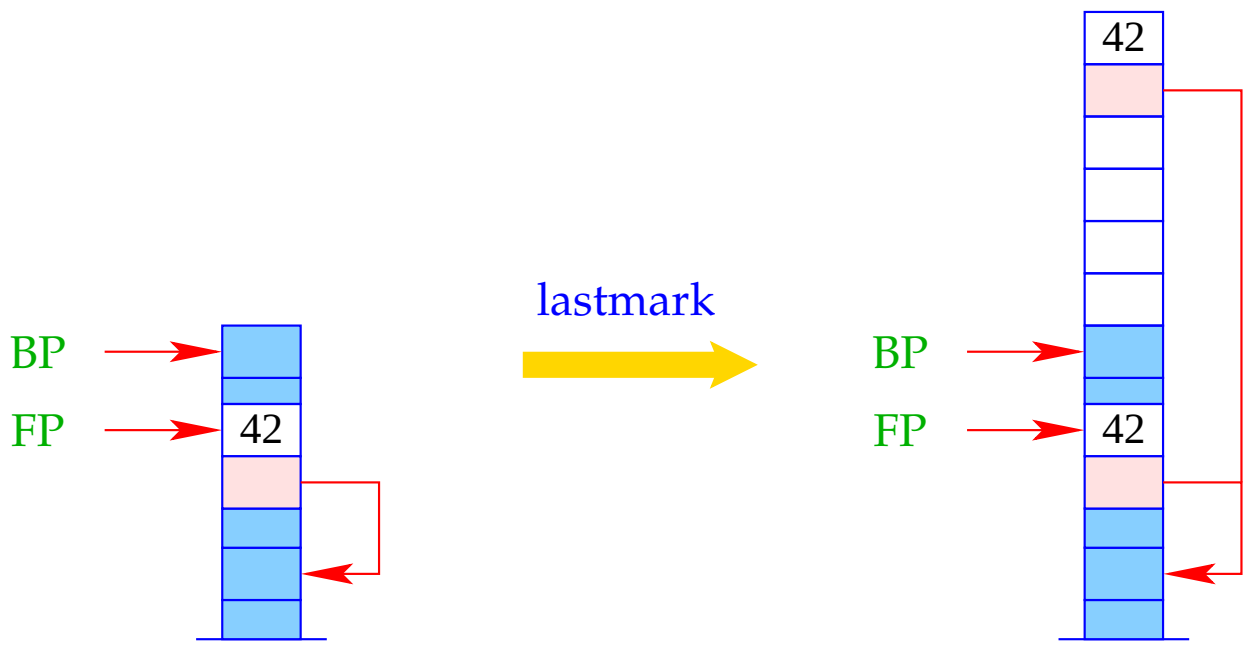
Consider a clause r: $p(X_1, \ldots, X_k) \leftarrow g_1, \ldots, g_n$
with m locals where $g_n \equiv q(t_1, \ldots, t_h)$. The interplay between code_C and code_G:

$$
\begin{aligned}
\text{code}_C\ r \quad = \quad & \text{pushenv m} \\
& \text{code}_G\ g_1\ \rho \\
& \ldots \\
& \text{code}_G\ g_{n-1}\ \rho \\
& \text{lastmark} \\
& \text{code}_A\ t_1\ \rho \\
& \ldots \\
& \text{code}_A\ t_h\ \rho \\
& \text{lastcall q/h m}
\end{aligned}
$$

Replacement:	mark B	$\Longrightarrow$	lastmark
	call q/h; popenv	$\Longrightarrow$	lastcall q/h m

If the current clause is not last or the $g_1, \ldots, g_{n-1}$ have created backtrack points, then FP $\leq$ BP :-)

Then lastmark creates a new frame but stores a reference to the predecessor:




```
if (FP ≤ BP) {
    SP = SP + 6;
    S[SP] = posCont; S[SP-1] = FPold;
}
```

If FP $>$ BP then lastmark does nothing :-)

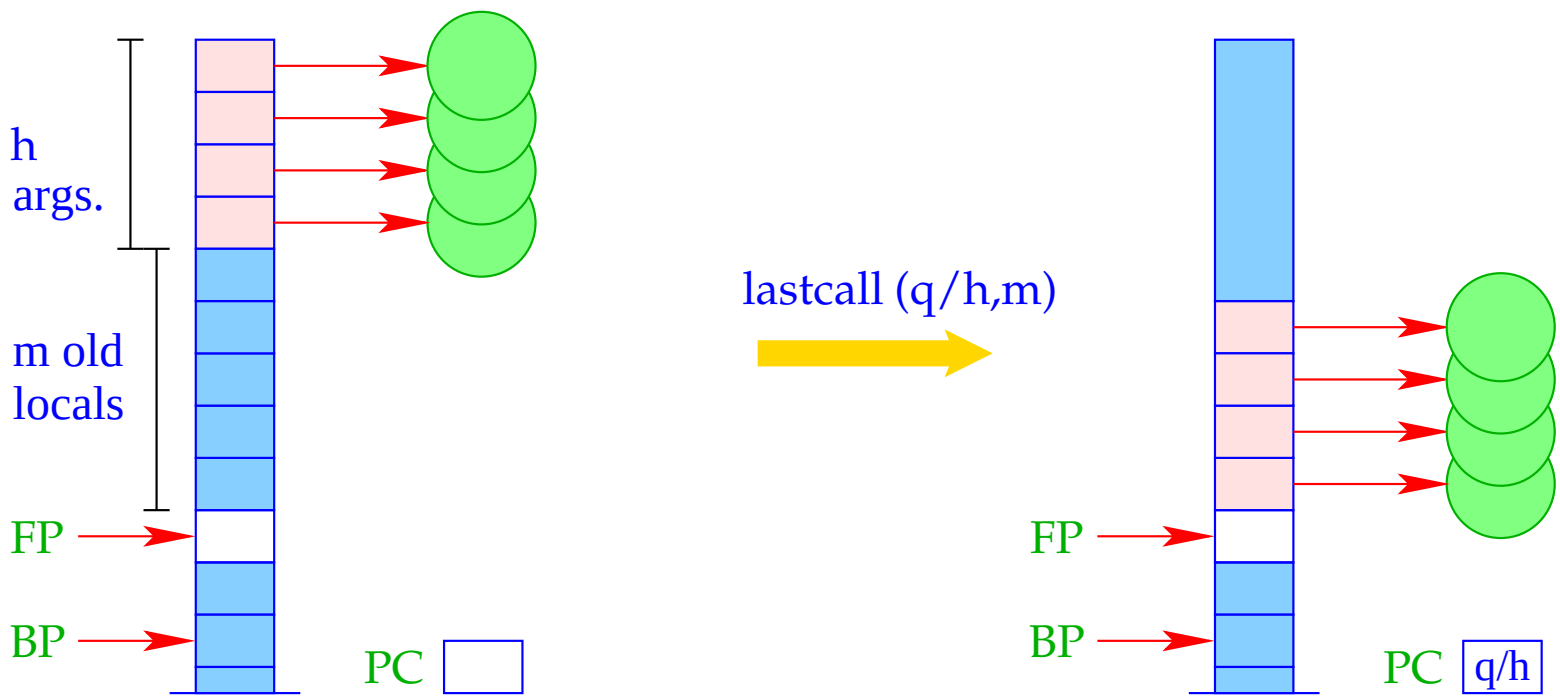
If FP $\leq$ BP, then lastcall q/h m behaves like a normal call q/h.

Otherwise, the current stack frame is re-used. This means that:

- the cells S[FP+1], S[FP+2], . . ., S[FP+h] receive the new values and
- q/h can be jumped to :-)

```
lastcall q/h m    =    if (FP ≤ BP)  call q/h;
                       else {
                          move m h;
                          jump q/h;
                       }
```

The difference between the old and the new addresses of the parameters m just equals the number of the local variables of the current clause :-))


h
args.
m old
locals
FP
BP
PC
lastcall (q/h,m)
FP
BP
PC
q/h

Example:

Consider the clause:

$$\mathsf{a}(X, Y) \leftarrow \mathsf{f}(\bar{X}, X_1), \mathsf{a}(\bar{X}_1, \bar{Y})$$

The last-call optimization for $\text{code}_C\ r$ yields:

```
                mark A        A:   lastmark
pushenv 3       putref 1           putref 3
                putvar 3           putref 2
                call f/2           lastcall a/2 3
```

Example:

Consider the clause:

$$\mathsf{a}(X, Y) \leftarrow \mathsf{f}(\bar{X}, X_1), \mathsf{a}(\bar{X}_1, \bar{Y})$$

The last-call optimization for $\text{code}_C\ r$ yields:

```
                mark A        A:   lastmark
pushenv 3       putref 1           putref 3
                putvar 3           putref 2
                call f/2           lastcall a/2 3
```

Note:

If the clause is last and the last literal is the only one, we can skip lastmark and can replace lastcall q/h m with the sequence move m n; jump p/n :-))

Example:

Consider the last clause of the app predicate:

$$\mathsf{app}(X, Y, Z) \quad \leftarrow \quad \bar{X} = [H|X'],\ \bar{Z} = [\bar{H}|Z'],\ \mathsf{app}(\bar{X}', \bar{Y}, \bar{Z}')$$

Here, the last call is the only one :-) Consequently, we obtain:

```
A:  pushenv 6
    putref 1
    ustruct [|]/2 B
    son 1
    uvar 4
    son 2
    uvar 5
    up C

B:  putvar 4
    putvar 5
    putstruct [|]/2
    bind

C:  putref 3
    ustruct [|]/2 D
    son 1
    uref 4
    son 2
    uvar 6
    up E

D:  check 4
    putref 4
    putvar 6
    putstruct [|]/2

E:  bind
    putref 5
    putref 2
    putref 6
    move 6 3
    jump app/3
```

35 Trimming of Stack Frames

Idea:

- Order local variables according to their life times;
- Pop the dead variables — if possible :-}

35 Trimming of Stack Frames

Idea:

- Order local variables according to their life times;
- Pop the dead variables — if possible :-}

Example:

Consider the clause:

$$\mathsf{a}(X, Z) \leftarrow \mathsf{p}_1(\bar{X}, X_1), \mathsf{p}_2(\bar{X}_1, X_2), \mathsf{p}_3(\bar{X}_2, X_3), \mathsf{p}_4(\bar{X}_3, \bar{Z})$$

35 Trimming of Stack Frames

Idea:

- Order local variables according to their life times;
- Pop the dead variables — if possible :-}

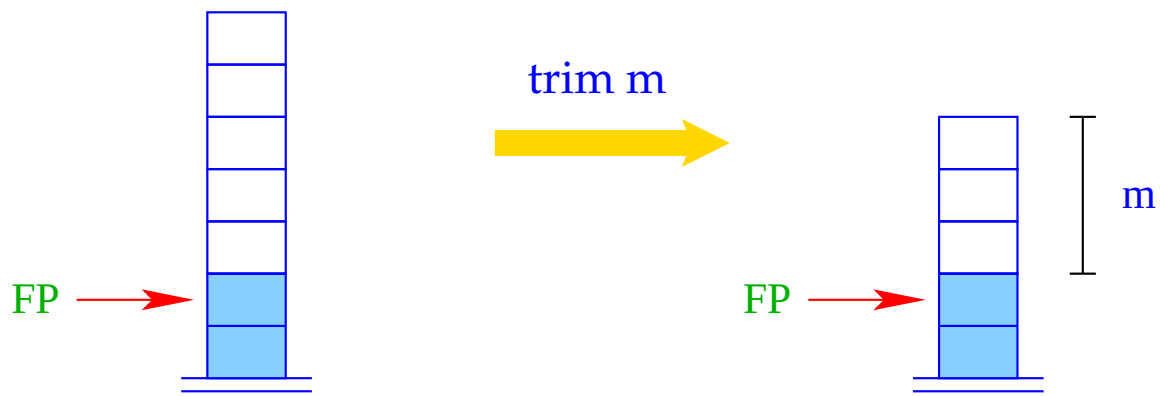
Example:

Consider the clause:

$$\mathsf{a}(X, Z) \leftarrow \mathsf{p}_1(\bar{X}, X_1), \mathsf{p}_2(\bar{X}_1, X_2), \mathsf{p}_3(\bar{X}_2, X_3), \mathsf{p}_4(\bar{X}_3, \bar{Z})$$

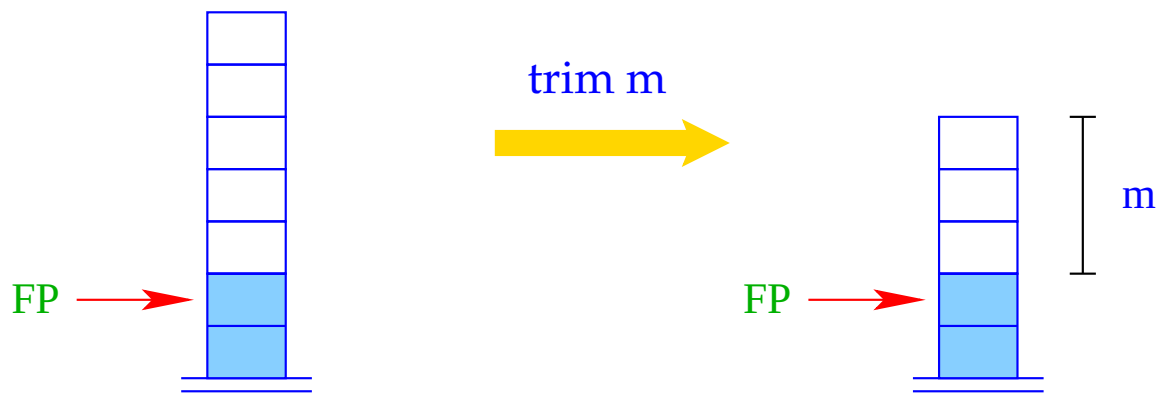
After the query $\mathsf{p}_2(\bar{X}_1, X_2)$, variable X_1 is dead.
After the query $\mathsf{p}_3(\bar{X}_2, X_3)$, variable X_2 is dead :-)

After every non-last goal with dead variables, we insert the instruction trim :



if (FP $\geq$ BP)
SP = FP + m;

After every non-last goal with dead variables, we insert the instruction trim :



if (FP $\geq$ BP)
SP = FP + m;

The dead locals can only be popped if no new backtrack point has been allocated :-)

Example (continued):

$$\mathsf{a}(X, Z) \leftarrow \mathsf{p}_1(\bar{X}, X_1), \mathsf{p}_2(\bar{X}_1, X_2), \mathsf{p}_3(\bar{X}_2, X_3), \mathsf{p}_4(\bar{X}_3, \bar{Z})$$

Ordering of the variables:

$$\rho = \{X \mapsto 1, Z \mapsto 2, X_3 \mapsto 3, X_2 \mapsto 4, X_1 \mapsto 5\}$$

The resulting code:

pushenv 5	A:	mark B		mark C	lastmark
mark A		putref 5		putref 4	putref 3
putref 1		putvar 4		putvar 3	putref 2
putvar 5		call $\mathsf{p}_2/2$		call $\mathsf{p}_3/2$	lastcall $\mathsf{p}_4/2$ 3
call $\mathsf{p}_1/2$	B:	trim 4	C:	trim 3	