Reinhard Wilhelm + Helmut Seidl

Abstract Machines

Saarbrücken + Trier

Summer 2002

0 Introduction

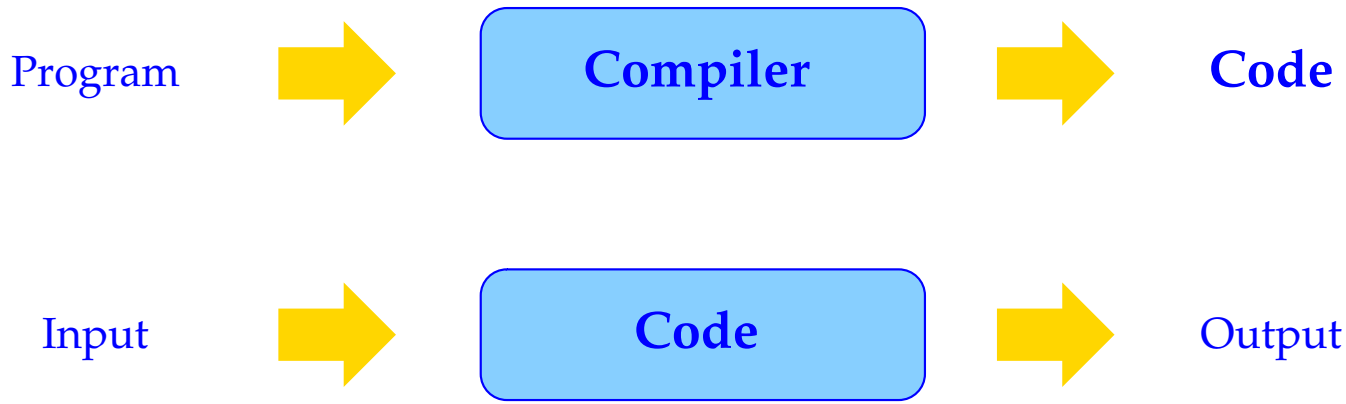
Principle of Interpretation:

Advantage: No precomputation on the program text $\Longrightarrow$ no/short startup-time

Disadvantages: Program parts are repeatedly analyzed during execution + less efficient access to program variables

$\Longrightarrow$ slower execution speed

Principle of Compilation:



Two Phases (at two different Times):

- Translation of the source program into a machine program (at compile time);
- Execution of the machine program on input data (at run time).

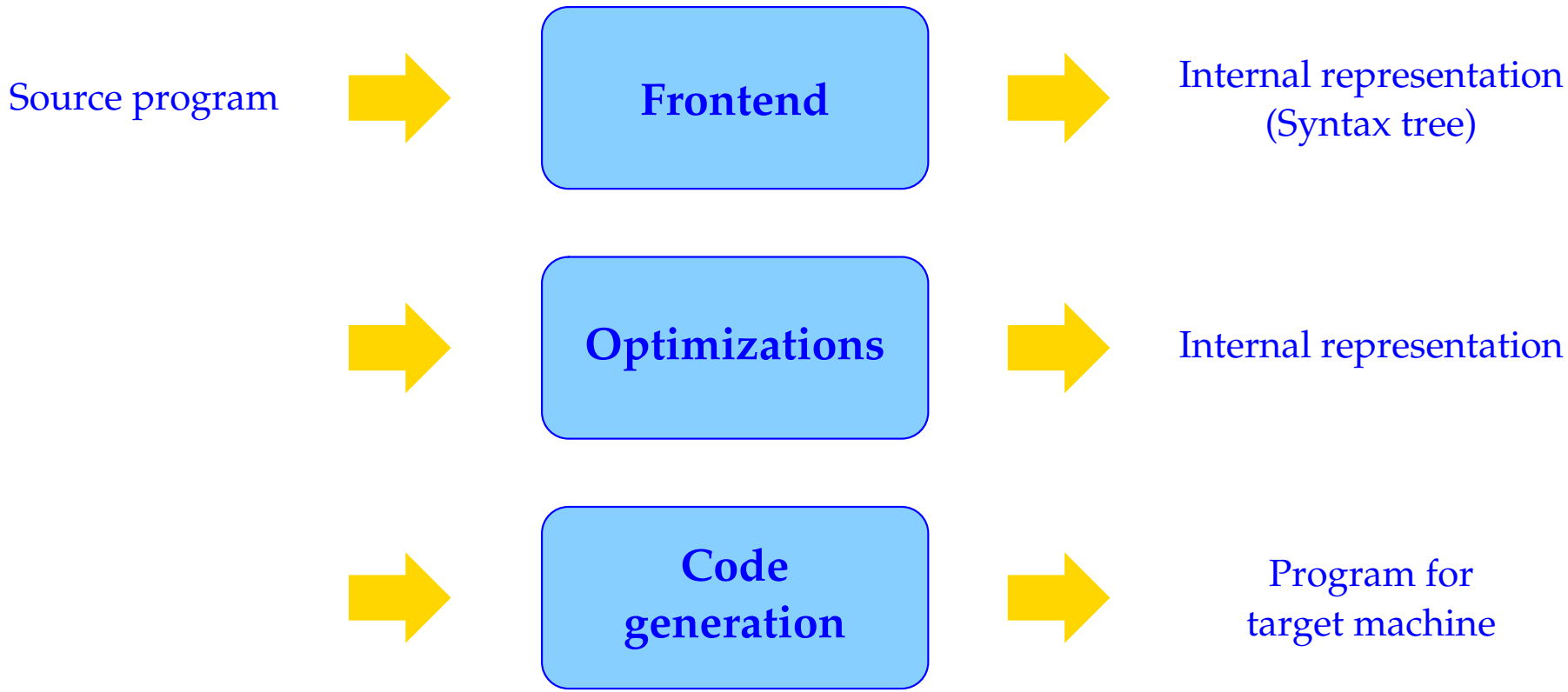
Preprocessing of the source program provides for

- efficient access to the values of program variables at run time
- global program transformations to increase execution speed.

Disadvantage: Compilation takes time

Advantage: Program execution is sped up $\Longrightarrow$ compilation pays off in long running or often run programs

Structure of a compiler:

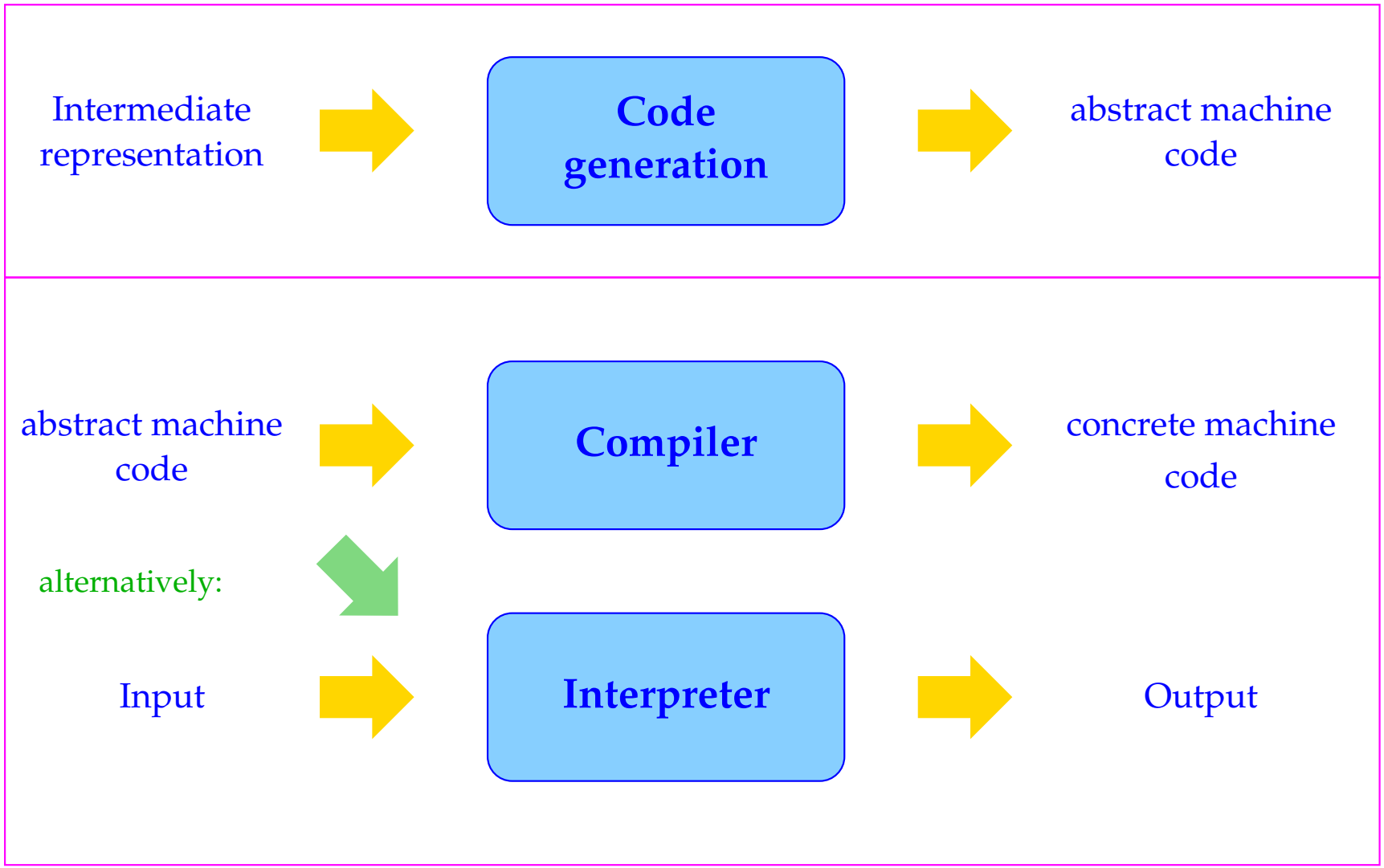


Subtasks in code generation:

Goal is a good exploitation of the hardware resources:

1. Instruction Selection: Selection of efficient, semantically equivalent instruction sequences;
2. Register-allocation: Best use of the available processor registers
3. Instruction Scheduling: Reordering of the instruction stream to exploit intra-processor parallelism

For several reasons, e.g. modularization of code generation and portability, code generation may be split into two phases:


Intermediate representation
Code generation
abstract machine code
abstract machine code
Compiler
concrete machine code
alternatively:
Input
Interpreter
Output

Abstract machine

- idealized architecture,
- simple code generation,
- easily implemented on real hardware.

Advantages:

- Porting the compiler to a new target architecture is simpler,
- Modularization makes the compiler easier to modify,
- Translation of program constructs is separated from the exploitation of architectural features.

Abstract machines for some programming languages:

Pascal	$\longrightarrow$	P-machine	
Smalltalk	$\longrightarrow$	Bytecode	
Prolog	$\longrightarrow$	WAM	("Warren Abstract Machine")
SML, Haskell	$\longrightarrow$	STGM	
Java	$\longrightarrow$	JVM	

We will consider the following languages and abstract machines:

C	$\longrightarrow$	CMa	//	*imperative*
PuF	$\longrightarrow$	MaMa	//	*functional*
Proll	$\longrightarrow$	WiM	//	*logic based*
multi-threaded C	$\longrightarrow$	threaded CMa	//	*concurrent*

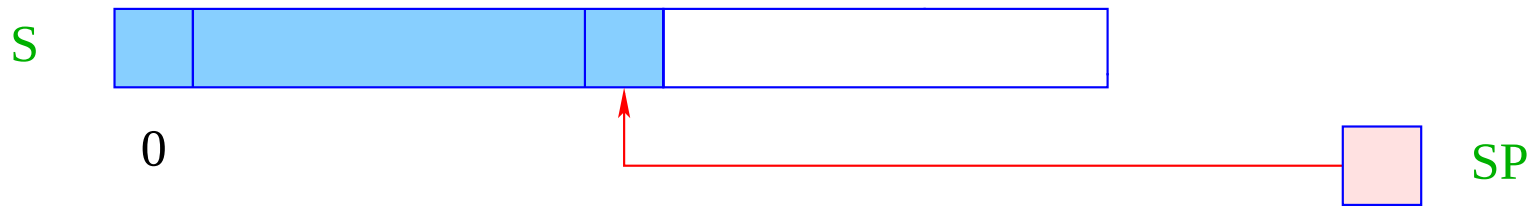
The Translation of C

1 The Architecture of the CMa

- Each abstract machine provides a set of instructions
- Instructions are executed on the abstract hardware
- This abstract hardware can be viewed as a set of data structures, which the instructions access
- ... and which are managed by the run-time system

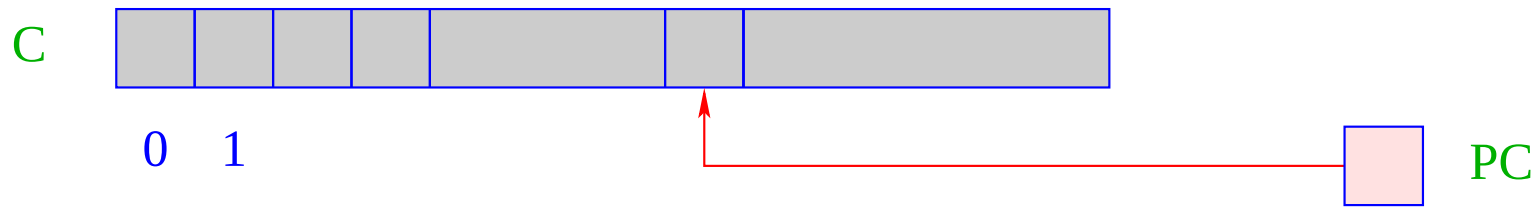
For the CMa we need:

The Data Store:



- S is the (data) store, onto which new cells are allocated in a LIFO discipline
  $\Longrightarrow$ Stack.
- SP ($\widehat{=}$ Stack Pointer) is a register, which contains the address of the topmost allocated cell,
  Simplification: All types of data fit into one cell of S.

The Code/Instruction Store:



- C is the Code store, which contains the program.

  Each cell of field C can store exactly one abstract instruction.

- PC ($\widehat{=}$ Program Counter) is a register, which contains the address of the instruction to be executed next.

- Initially, PC contains the address 0.

  $\Longrightarrow$ C[0] contains the instruction to be executed first.

Execution of Programs:

- The machine loads the instruction in C[PC] into a Instruction-Register IR and executes it
- PC is incremented by 1 before the execution of the instruction

```
while (true) {
        IR = C[PC]; PC++;
        execute (IR);
}
```

- The execution of the instruction may overwrite the PC (jumps).
- The Main Cycle of the machine will be halted by executing the instruction halt , which returns control to the environment, e.g. the operating system
- More instructions will be introduced by demand

2 Simple expressions and assignments

Problem: evaluate the expression $(1 + 7) * 3$!

This means: generate an instruction sequence, which

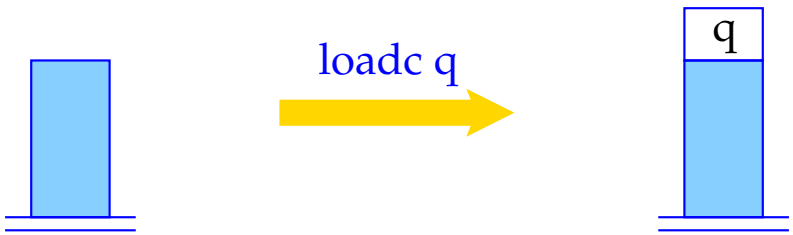
- determines the value of the expression and
- pushes it on top of the stack...

Idea:

- first compute the values of the subexpressions,
- save these values on top of the stack,
- then apply the operator.

The general principle:

- instructions expect their arguments on top of the stack,
- execution of an instruction consumes its operands,
- results, if any, are stored on top of the stack.

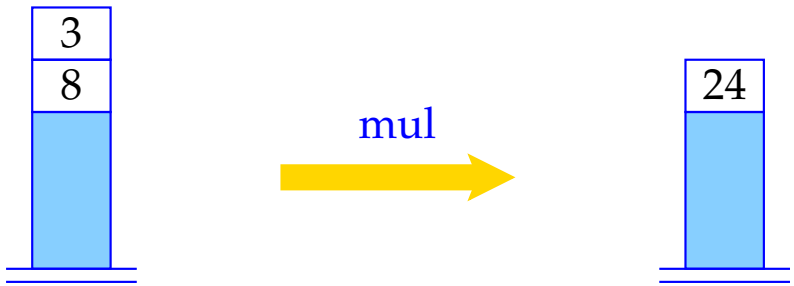


SP++;

S[SP] = q;

Instruction loadc q needs no operand on top of the stack, pushes the constant q onto the stack.

Note: the content of register SP is only implicitly represented, namely through the height of the stack.



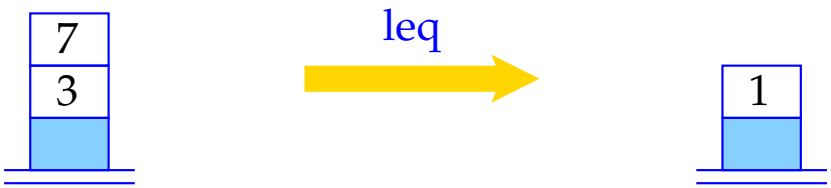
SP--;

S[SP] = S[SP] ∗ S[SP+1];

mul expects two operands on top of the stack, consumes both, and pushes their product onto the stack.

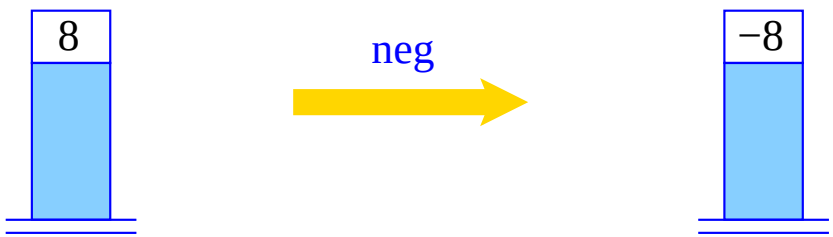
... the other binary arithmetic and logical instructions, add, sub, div, mod, and, or and xor, work analogously, as do the comparison instructions eq, neq, le, leq, gr and geq.

Example: The operator leq



Remark: 0 represents *false*, all other integers *true*.

Unary operators neg and not consume one operand and produce one result.

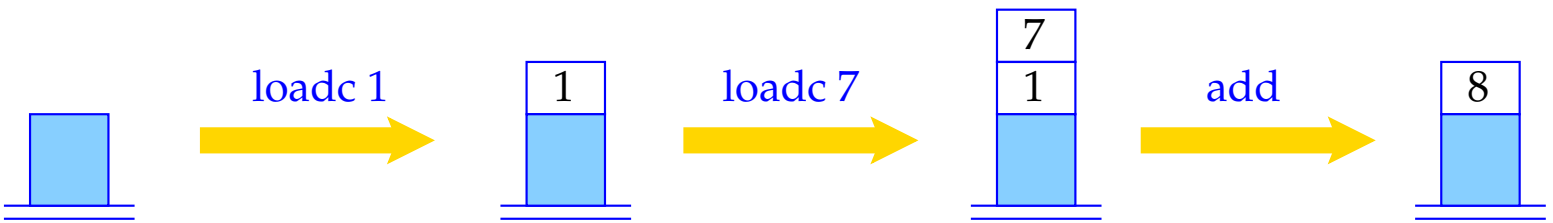


S[SP] = – S[SP];

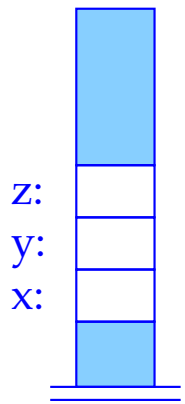
Example: Code for $1 + 7$:

loadc 1 loadc 7 add

Execution of this code sequence:



Variables are associated with cells in S:



Code generation will be described by some Translation Functions, code, code_L, and code_R.

Arguments: A program construct and a function ρ. ρ delivers for each variable x the relative address of x. ρ is called Address Environment.

Variables can be used in two different ways:

Example: $x = y + 1$

We are interested in the value of y, but in the address of x.

The syntactic position determines, whether the L-value or the R-value of a variable is required.

L-value of x = address of x

R-value of x = content of x

$\text{code}_R\ e\ \rho$	produces code to compute the R-value of e in the address environment ρ
$\text{code}_L\ e\ \rho$	analogously for the L-value

Note:

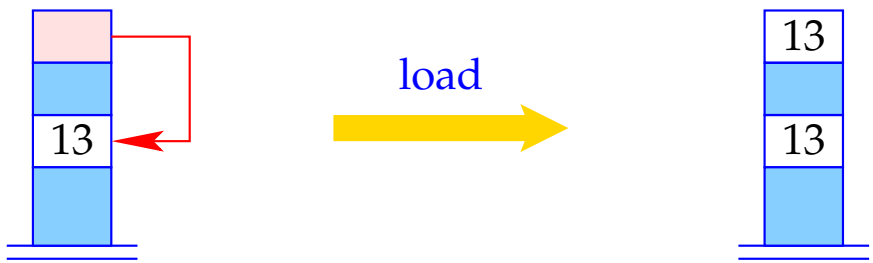
Not every expression has an L-value (Ex.: $x + 1$).

We define:

$$
\begin{array}{lcl}
\text{code}_\text{R}\ (e_1 + e_2)\ \rho & = & \text{code}_\text{R}\ e_1\ \rho \\
 & & \text{code}_\text{R}\ e_2\ \rho \\
 & & \text{add} \\
 & & \quad \text{... analogously for the other binary operators} \\
\text{code}_\text{R}\ (-e)\ \rho & = & \text{code}_\text{R}\ e\ \rho \\
 & & \text{neg} \\
 & & \quad \text{... analogously for the other unary operators} \\
\text{code}_\text{R}\ q\ \rho & = & \text{loadc q} \\
\text{code}_\text{L}\ x\ \rho & = & \text{loadc}\ (\rho\, x) \\
 & & \ldots
\end{array}
$$

$$\begin{array}{lcl} \text{code}_R\ x\ \rho & = & \text{code}_L\ x\ \rho \\ & & \texttt{load} \end{array}$$

The instruction `load` loads the contents of the cell, whose address is on top of the stack.

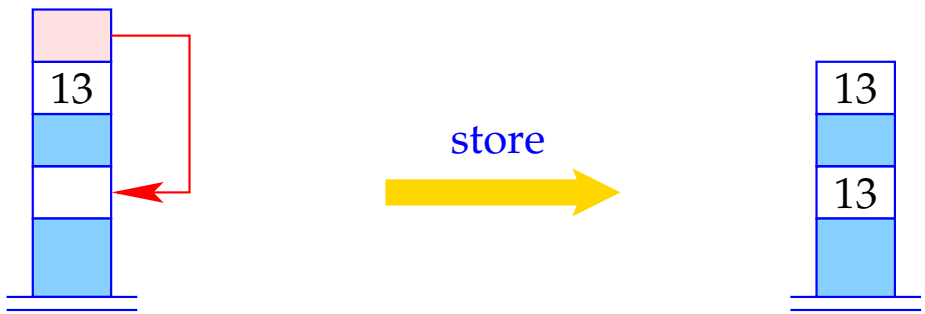


S[SP] = S[S[SP]];

$$\begin{aligned} \text{code}_R\ (x = e)\ \rho \quad = \quad & \text{code}_R\ e\ \rho \\ & \text{code}_L\ x\ \rho \\ & \text{store} \end{aligned}$$

store writes the contents of the second topmost stack cell into the cell, whose address in on top of the stack, and leaves the written value on top of the stack.

Note: this is different from the corresponding store–instruction of the P–machine in Wilhelm/Maurer!



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

SP--;

Example:

Code for $e \equiv x = y - 1$ with $\rho = \{x \mapsto 4, y \mapsto 7\}$.

$\text{code}_R\ e\ \rho$ produces:

```
loadc 7
load
```

```
loadc 1
sub
```

```
loadc 4
store
```

Improvements:

Introduction of special instructions for frequently used instruction sequences, e.g.,

```
loada q    =    loadc q
                load

storea q   =    loadc q
                store
```

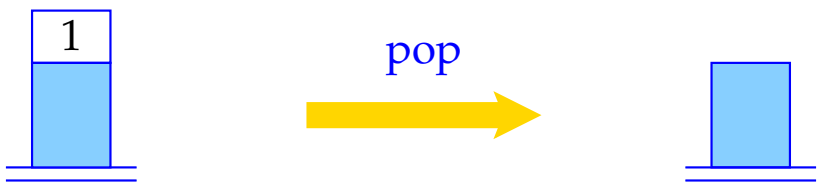
3 Statements and Statement Sequences

Is e an expression, then $e;$ is a statement.

Statements do not deliver a value. The contents of the SP before and after the execution of the generated code must therefore be the same.

$$\begin{aligned} \text{code } e;\ \rho \quad &= \quad \text{code}_R\ e\ \rho \\ &\qquad\quad \text{pop} \end{aligned}$$

The instruction pop eliminates the top element of the stack.



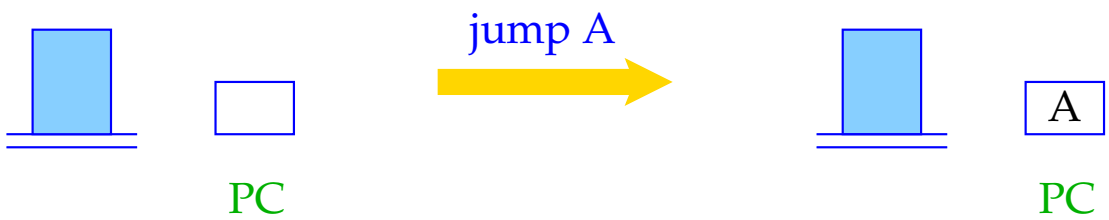
SP--;

The code for a statement sequence is the concatenation of the code for the statements of the sequence:

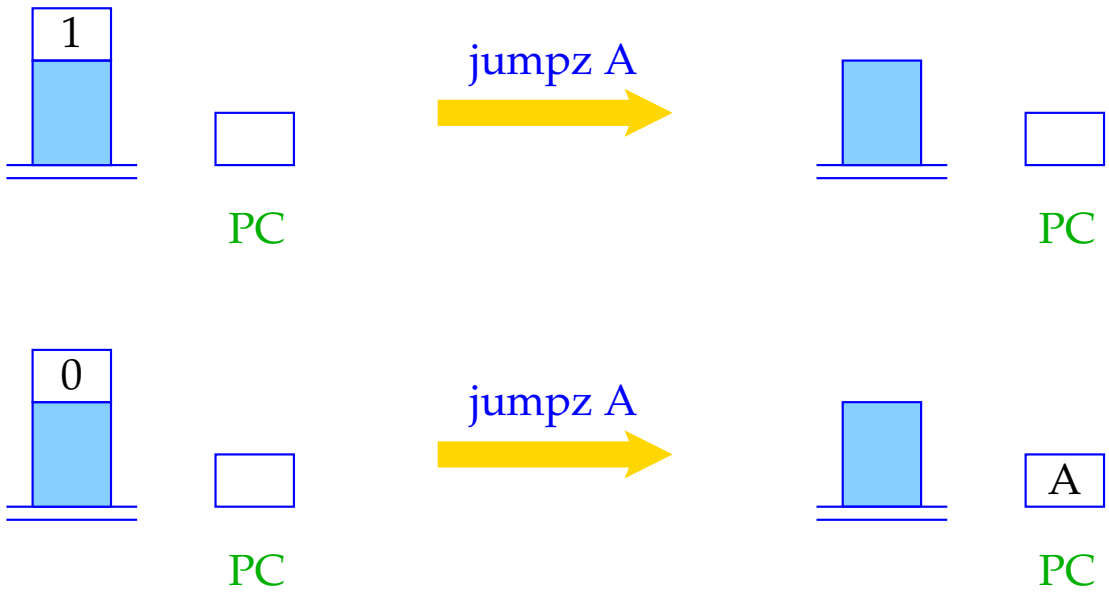
$$
\begin{array}{lcll}
\text{code}\ (s\ ss)\ \rho & = & \text{code}\ s\ \rho & \\
 & & \text{code}\ ss\ \rho & \\
\text{code}\ \varepsilon\ \rho & = & // & \textit{empty sequence of instructions}
\end{array}
$$

4 Conditional and Iterative Statements

We need jumps to deviate from the serial execution of consecutive statements:



PC = A;



if (S[SP] == 0) PC = A;

SP--;

For ease of comprehension, we use symbolic jump targets. They will later be replaced by absolute addresses.

Instead of absolute code addresses, one could generate relative addresses, i.e., relative to the actual PC.

Advantages:

- smaller addresses suffice most of the time;
- the code becomes relocatable, i.e., can be moved around in memory.

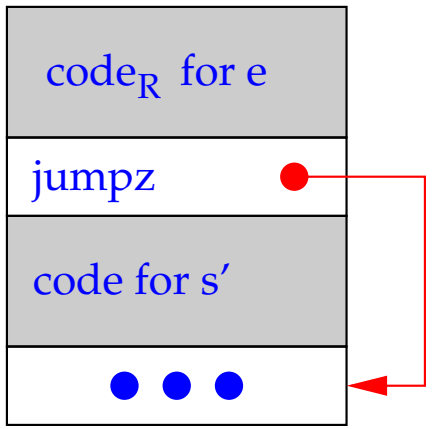
4.1 One-sided Conditional Statement

Let us first regard $s \equiv \textbf{if}\ (e)\ s'$.

Idea:

- Put code for the evaluation of e and s' consecutively in the code store,
- Insert a conditional jump (jump on zero) in between.

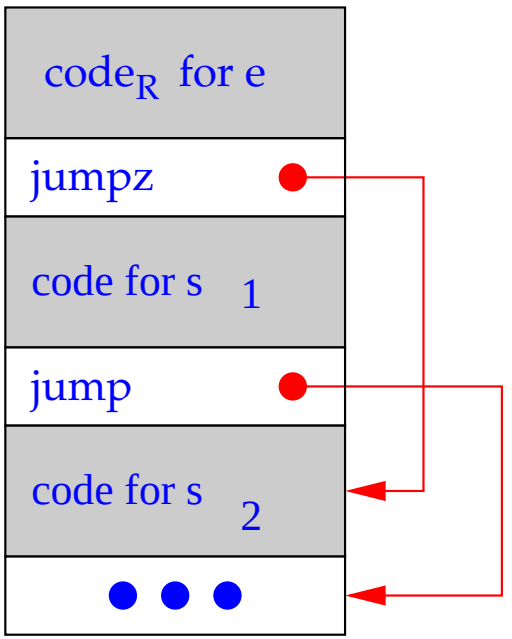
$$
\begin{array}{lcl}
\text{code } s\ \rho & = & \text{code}_R\ e\ \rho \\
 & & \text{jumpz A} \\
 & & \text{code } s'\ \rho \\
 & \text{A}: & \ldots
\end{array}
$$



4.2 Two-sided Conditional Statement

Let us now regard $s \equiv$ **if** (e) s_1 **else** s_2. The same strategy yields:

$$
\begin{array}{rcl}
\text{code } s\ \rho & = & \text{code}_R\ e\ \rho \\
 & & \text{jumpz A} \\
 & & \text{code } s_1\ \rho \\
 & & \text{jump B} \\
\text{A}: & & \text{code } s_2\ \rho \\
\text{B}: & & \ldots
\end{array}
$$



Example: Be $\rho = \{x \mapsto 4, y \mapsto 7\}$ and

$$
\begin{array}{lll}
s \equiv & \textbf{if}\,(x > y) & (i) \\
& \quad x = x - y; & (ii) \\
& \textbf{else}\; y = y - x; & (iii)
\end{array}
$$

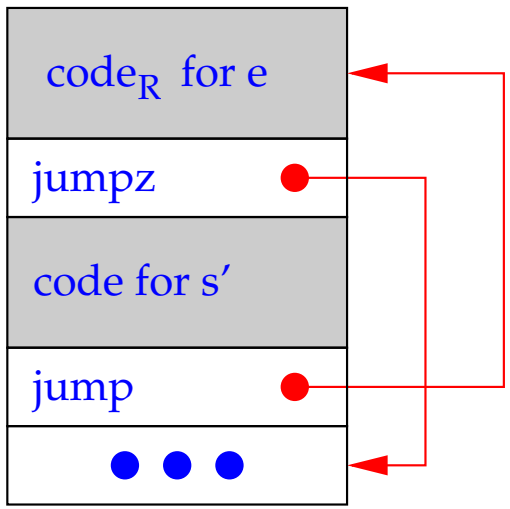
code s ρ produces:

(i)	(ii)		(iii)
loada 4	loada 4	A:	loada 7
loada 7	loada 7		loada 4
ge	sub		sub
jumpz A	storea 4		storea 7
	pop		pop
	jump B	B:	...

4.3 while-Loops

Let us regard the loop $s \equiv \textbf{while}\ (e)\ s'$. We generate:

$$
\begin{array}{lll}
\text{code } s\ \rho & = & \\
 & \text{A}: & \text{code}_\text{R}\ e\ \rho \\
 & & \text{jumpz B} \\
 & & \text{code } s'\ \rho \\
 & & \text{jump A} \\
 & \text{B}: & \ldots
\end{array}
$$



Example: Be $\rho = \{a \mapsto 7, b \mapsto 8, c \mapsto 9\}$ and s the statement:

$$\textbf{while}\ (a > 0)\ \{c = c + 1;\ a = a - b;\}$$

code $s\ \rho$ produces the sequence:

```
A:  loada 7     loada 9     loada 7     B:  ...
    loadc 0     loadc 1     loada 8
    ge          add         sub
    jumpz B     storea 9    storea 7
                pop         pop
                            jump A
```

4.4 for-Loops

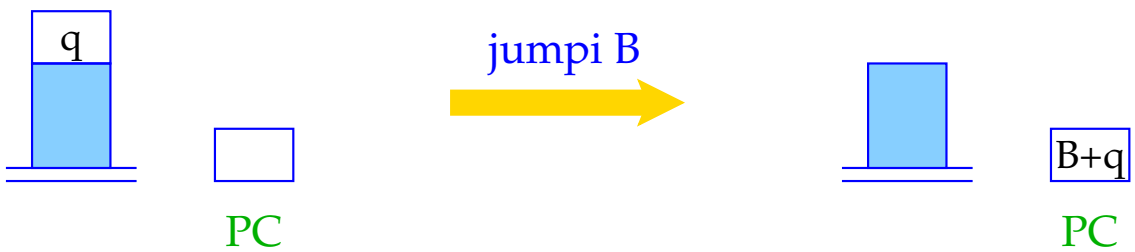
The **for**-loop $s \equiv \textbf{for}\ (e_1; e_2; e_3)\ s'$ is equivalent to the statement sequence $e_1;\ \textbf{while}\ (e_2)\ \{s'\ e_3;\}$ – provided that s' contains no **continue**-statement.
We therefore translate:

$$
\begin{array}{rcl}
\text{code}\ s\ \rho & = & \text{code}_\text{R}\ e_1 \\
 & & \text{pop} \\
 & \text{A}: & \text{code}_\text{R}\ e_2\ \rho \\
 & & \text{jumpz B} \\
 & & \text{code}\ s'\ \rho \\
 & & \text{code}_\text{R}\ e_3\ \rho \\
 & & \text{pop} \\
 & & \text{jump A} \\
 & \text{B}: & \ldots
\end{array}
$$

4.5 The switch-Statement

Idea:

- Multi-target branching in constant time!
- Use a jump table, which contains at its i-th position the jump to the beginning of the i-th alternative.
- Realized by indexed jumps.



PC = B + S[SP];

SP--;

Simplification:

We only regard **switch**-statements of the following form:

$$
\begin{array}{lcl}
s & \equiv & \textbf{switch}\ (e)\ \{ \\
 & & \quad \textbf{case}\ 0\text{:} \quad ss_0\ \textbf{break;} \\
 & & \quad \textbf{case}\ 1\text{:} \quad ss_1\ \textbf{break;} \\
 & & \quad \vdots \\
 & & \quad \textbf{case}\ k-1\text{:} \quad ss_{k-1}\ \textbf{break;} \\
 & & \quad \textbf{default:} \quad ss_k \\
 & & \}
\end{array}
$$

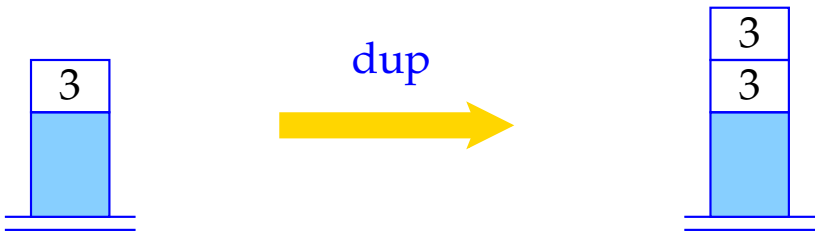
s is then translated into the instruction sequence:

$$
\begin{array}{lcllll}
\text{code } s\ \rho & = & \text{code}_R\ e\ \rho & \mathrm{C}_0\text{:} & \text{code } ss_0\ \rho & \mathrm{B}\text{:} \quad \text{jump } \mathrm{C}_0 \\
 & & \text{check } 0\ k\ \mathrm{B} & & \text{jump D} & \quad \ldots \\
 & & & & \ldots & \quad \text{jump } \mathrm{C}_k \\
 & & & \mathrm{C}_k\text{:} & \text{code } ss_k\ \rho & \mathrm{D}\text{:} \quad \ldots \\
 & & & & \text{jump D} &
\end{array}
$$

- The Macro check $0\ k$ B checks, whether the R-value of e is in the interval $[0, k]$, and executes an indexed jump into the table B
- The jump table contains direct jumps to the respective alternatives.
- At the end of each alternative is an unconditional jump out of the **switch**-statement.

check 0 *k* B	=	dup	dup		jumpi B
		loadc 0	loadc k	A:	pop
		geq	le		loadc k
		jumpz A	jumpz A		jumpi B

- The R-value of *e* is still needed for indexing after the comparison. It is therefore copied before the comparison.
- This is done by the instruction dup.
- The R-value of *e* is replaced by *k* before the indexed jump is executed if it is less than 0 or greater than *k*.



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

SP++;

Note:

- The jump table could be placed directly after the code for the Macro check. This would save a few unconditional jumps. However, it may require to search the **switch**-statement twice.
- If the table starts with u instead of 0, we have to decrease the R-value of e by u before using it as an index.
- If all potential values of e are definitely in the interval $[0, k]$, the macro check is not needed.

5 Storage Allocation for Variables

Goal:

Associate statically, i.e. at compile time, with each variable x a fixed (relative) address $\rho\, x$

Assumptions:

- variables of basic types, e.g. **int**, ... occupy one storage cell.
- variables are allocated in the store in the order, in which they are declared, starting at address 1.

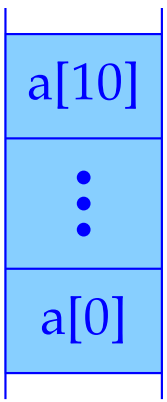
Consequently, we obtain for the declaration $d \equiv t_1\ x_1;\ \ldots\ t_k\ x_k;$ (t_i basic type) the address environment ρ such that

$$\rho\, x_i = i, \quad i = 1, \ldots, k$$

5.1 Arrays

Example: **int** [11] a;

The array a consists of 11 components and therefore needs 11 cells.
$\rho\, a$ is the address of the component $a[0]$.



We need a function sizeof (notation: $|\cdot|$), computing the space requirement of a type:

$$|t| = \begin{cases} 1 & \text{if } t \text{ basic} \\ k \cdot |t'| & \text{if } t \equiv t'[k] \end{cases}$$

Accordingly, we obtain for the declaration $d \equiv t_1\ x_1; \ldots t_k\ x_k;$

$$\begin{array}{lcll} \rho\, x_1 & = & 1 & \\ \rho\, x_i & = & \rho\, x_{i-1} + |t_{i-1}| & \text{for } i > 1 \end{array}$$

Since $|\cdot|$ can be computed at compile time, also ρ can be computed at compile time.

Task:

Extend code_L and code_R to expressions with accesses to array components.

Be $t[c]\ a;$ the declaration of an array a.

To determine the start address of a component $a[i]$, we compute
$\rho\, a + |t| * (\textit{R-value of } i)$.

In consequence:

$$
\begin{array}{lcl}
\text{code}_L\ a[e]\ \rho & = & \text{loadc}\ (\rho\, a) \\
 & & \text{code}_R\ e\ \rho \\
 & & \text{loadc}\ |t| \\
 & & \text{mul} \\
 & & \text{add}
\end{array}
$$

... or more general:

$$\begin{aligned} \text{code}_\text{L}\ e_1[e_2]\ \rho \quad = \quad & \text{code}_\text{R}\ e_1\ \rho \\ & \text{code}_\text{R}\ e_2\ \rho \\ & \text{loadc}\ |t| \\ & \text{mul} \\ & \text{add} \end{aligned}$$

Remark:

- In C, an array is a pointer. A declared array a is a pointer-constant, whose R-value is the start address of the array.
- Formally, we define for an array e: $\text{code}_\text{R}\ e\ \rho = \text{code}_\text{L}\ e\ \rho$
- In C, the following are equivalent (as L-values):

$$2[a] \qquad a[2] \qquad a+2$$

Normalization: Array names and expressions evaluating to arrays occur in front of index brackets, index expressions inside the index brackets.

5.2 Structures

In Modula and Pascal, structures are called Records.

Simplification:

Names of structure components are not used elsewhere.
Alternatively, one could manage a separate environment ρ_{st} for each structure type st.

Be **struct** { **int** a; **int** b; } x; part of a declaration list.

- x has as relative address the address of the first cell allocated for the structure.
- The components have addresses relative to the start address of the structure. In the example, these are $a \mapsto 0$, $b \mapsto 1$.

Let $t \equiv \textbf{struct}\,\{t_1\ c_1; \ldots t_k\ c_k;\}$. We have

$$\begin{aligned} |t| &= \sum_{i=1}^{k} |t_i| \\ \rho\, c_1 &= 0 \quad \text{and} \\ \rho\, c_i &= \rho\, c_{i-1} + |t_{i-1}| \quad \text{for } i > 1 \end{aligned}$$

We thus obtain:

$$\begin{aligned} \text{code}_\text{L}\ (e.c)\ \rho \quad = \quad & \text{code}_\text{L}\ e\ \rho \\ & \text{loadc}\ (\rho\, c) \\ & \text{add} \end{aligned}$$

Example:

Be **struct** { **int** a; **int** b; } x; such that $\rho = \{x \mapsto 13, a \mapsto 0, b \mapsto 1\}$.

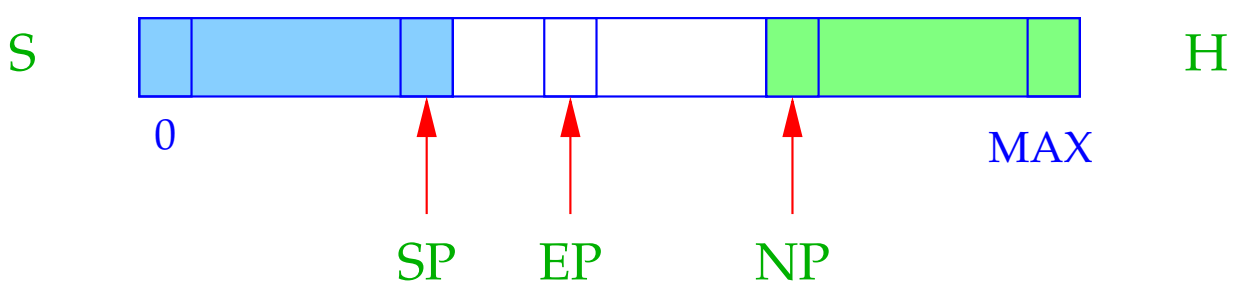
This yields:

$$\begin{array}{lll} \text{code}_{\text{L}}\ (x.b)\ \rho & = & \text{loadc } 13 \\ & & \text{loadc } 1 \\ & & \text{add} \end{array}$$

6 Pointer and Dynamic Storage Management

Pointer allow the access to anonymous, dynamically generated objects, whose life time is not subject to the LIFO-principle.

$\Longrightarrow$ We need another potentially unbounded storage area H – the Heap.



NP $\widehat{=}$ New Pointer; points to the lowest occupied heap cell.

EP $\widehat{=}$ Extreme Pointer; points to the uppermost cell, to which SP can point (during execution of the actual function).

Idea:

- Stack and Heap grow toward each other in S, but must not collide. (Stack Overflow).
- A collision may be caused by an increment of SP or a decrement of NP.
- EP saves us the check for collision at the stack operations.
- The checks at heap allocations are still necessary.

What can we do with pointers (pointer values)?

- set a pointer to a storage cell,
- dereference a pointer, access the value in a storage cell pointed to by a pointer.

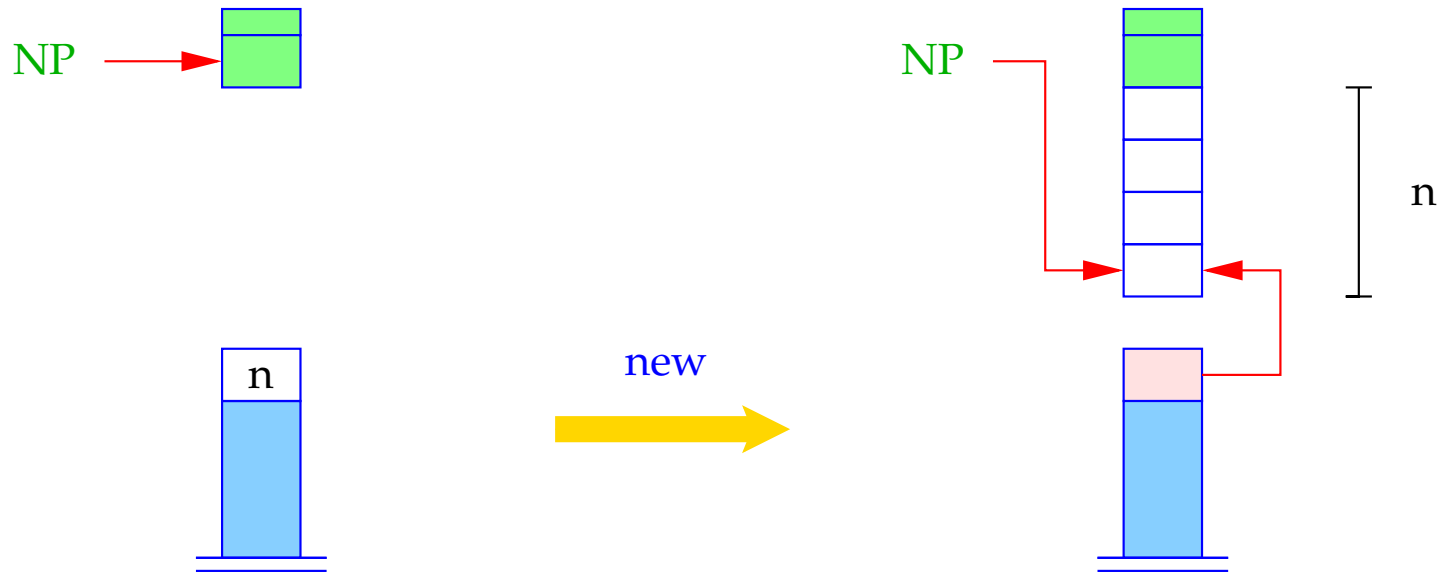
There a two ways to set a pointer:

(1) A call **malloc** (e) reserves a heap area of the size of the value of e and returns a pointer to this area:

$$\begin{aligned} \text{code}_\text{R}\ \textbf{malloc}\,(e)\ \rho \quad &= \quad \text{code}_\text{R}\ e\ \rho \\ & \qquad\ \ \text{new} \end{aligned}$$

(2) The application of the address operator & to a variable returns a pointer to this variable, i.e. its address ($\widehat{=}$ L-value). Therefore:

$$\text{code}_\text{R}\ (\&e)\ \rho = \text{code}_\text{L}\ e\ \rho$$




```
if (NP - S[SP] ≤ EP)
        S[SP] = NULL;
else {
        NP = NP - S[SP];
        S[SP] = NP;
}
```

- NULL is a special pointer constant, identified with the integer constant 0.
- In the case of a collision of stack and heap the NULL-pointer is returned.

Dereferencing of Pointers:

The application of the operator $*$ to the expression e returns the contents of the storage cell, whose address is the R-value of e:

$$\text{code}_\text{L}\ (*e)\ \rho = \text{code}_\text{R}\ e\ \rho$$

Example:

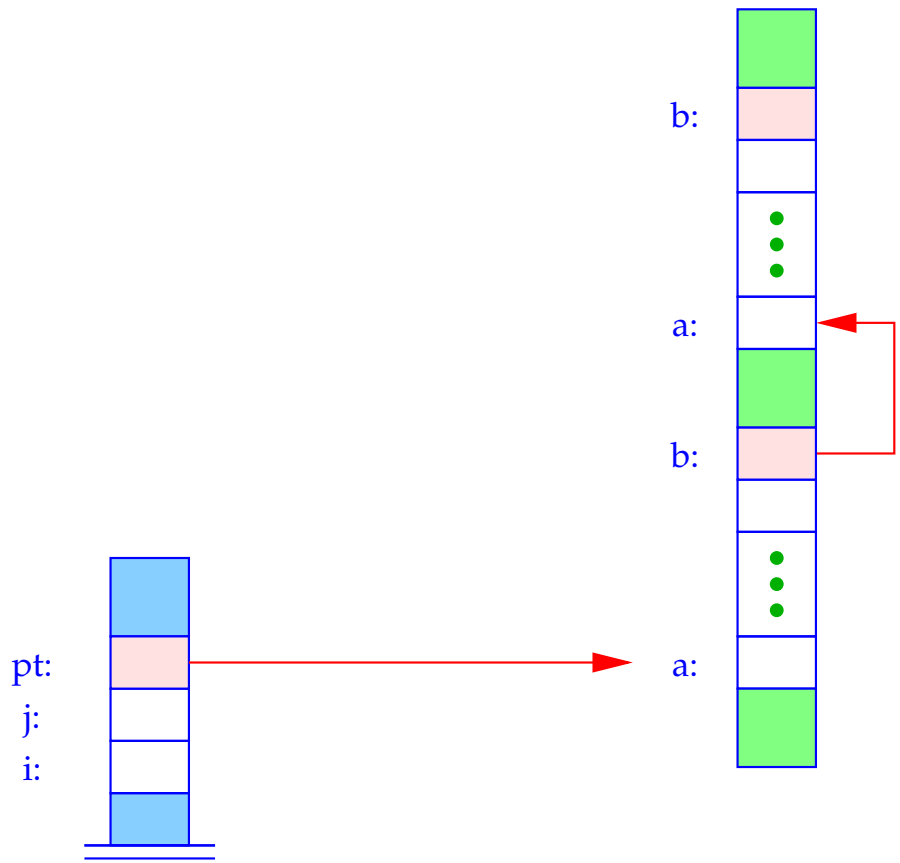
Given the declarations

struct t { **int** $a[7]$; **struct** t $*b$; };
int i, j;
struct t $*pt$;

and the expression $((pt \rightarrow b) \rightarrow a)[i+1]$

Because of $e \rightarrow a \equiv (*e).a$ holds:

$$\begin{aligned} \text{code}_\text{L}\ (e \rightarrow a)\ \rho \quad = \quad & \text{code}_\text{R}\ e\ \rho \\ & \text{loadc}\ (\rho\, a) \\ & \text{add} \end{aligned}$$


b:
a:
b:
a:
pt:
j:
i:

Be $\rho = \{i \mapsto 1, j \mapsto 2, pt \mapsto 3, a \mapsto 0, b \mapsto 7\}$. Then:

$\text{code}_L\ ((pt \to b) \to a)[i+1]\ \rho$

$=$ $\text{code}_R\ ((pt \to b) \to a)\ \rho$
$\text{code}_R\ (i+1)\ \rho$
loadc 1
mul
add

$=$ $\text{code}_R\ ((pt \to b) \to a)\ \rho$
loada 1
loadc 1
add
loadc 1
mul
add

For arrays, their R-value equals their L-value. Therefore:

$$\begin{array}{lcll} \text{code}_\text{R}\ ((pt \to b) \to a)\ \rho & = & \text{code}_\text{R}\ (pt \to b)\ \rho & = \text{loada 3} \\ & & \text{loadc 0} & \text{loadc 7} \\ & & \text{add} & \text{add} \\ & & & \text{load} \\ & & & \text{loadc 0} \\ & & & \text{add} \end{array}$$

In total, we obtain the instruction sequence:

loada 3	load	loada 1	loadc 1
loadc 7	loadc 0	loadc 1	mul
add	add	add	add

7 Conclusion

We tabulate the cases of the translation of expressions:

$$
\begin{array}{lcll}
\text{code}_\text{L}\ (e_1[e_2])\ \rho & = & \text{code}_\text{R}\ e_1\ \rho & \\
 & & \text{code}_\text{R}\ e_2\ \rho & \\
 & & \text{loadc}\ |t| & \\
 & & \text{mul} & \\
 & & \text{add} & \text{if } e_1 \text{ has type } t* \text{ or } t[] \\
 & & & \\
\text{code}_\text{L}\ (e.a)\ \rho & = & \text{code}_\text{L}\ e\ \rho & \\
 & & \text{loadc}\ (\rho\, a) & \\
 & & \text{add} &
\end{array}
$$

$$
\begin{array}{lcll}
\text{code}_\text{L}\ (*e)\ \rho & = & \text{code}_\text{R}\ e\ \rho & \\
\text{code}_\text{L}\ x\ \rho & = & \text{loadc}\ (\rho\, x) & \\
\text{code}_\text{R}\ (\&e)\ \rho & = & \text{code}_\text{L}\ e\ \rho & \\
\text{code}_\text{R}\ e\ \rho & = & \text{code}_\text{L}\ e\ \rho & \text{if } e \text{ is an array} \\
\text{code}_\text{R}\ (e_1 \,\square\, e_2)\ \rho & = & \text{code}_\text{R}\ e_1\ \rho & \\
 & & \text{code}_\text{R}\ e_2\ \rho & \\
 & & \text{op} & \text{op instruction for operator `}\square\text{'}
\end{array}
$$

$$
\begin{array}{lcll}
\text{code}_R\ q\ \rho & = & \text{loadc q} & q \text{ constant} \\
\\
\text{code}_R\ (e_1 = e_2)\ \rho & = & \text{code}_R\ e_2\ \rho & \\
 & & \text{code}_L\ e_1\ \rho & \\
 & & \text{store} & \\
\\
\text{code}_R\ e\ \rho & = & \text{code}_L\ e\ \rho & \\
 & & \text{load} & \text{otherwise}
\end{array}
$$

Example: **int** $a[10],\ *b;$ with $\rho = \{a \mapsto 7, b \mapsto 17\}$.

For the statement: $*a = 5;$ we obtain:

$$\begin{array}{lcl} \text{code}_L\ (*a)\ \rho & = & \text{code}_R\ a\ \rho \quad = \quad \text{code}_L\ a\ \rho \quad = \quad \text{loadc } 7 \\ \text{code}\ (*a = 5;)\ \rho & = & \text{loadc } 5 \\ & & \text{loadc } 7 \\ & & \text{store} \\ & & \text{pop} \end{array}$$

As an exercise translate:

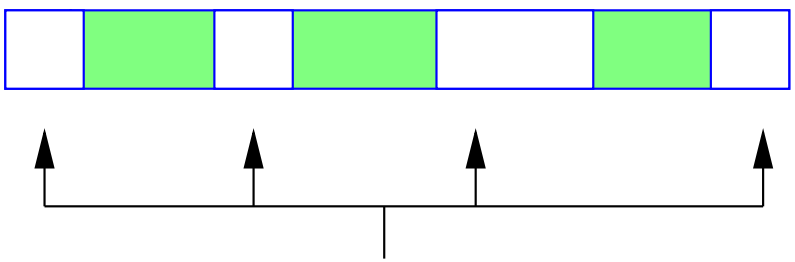
$$s_1 \equiv b = (\&a) + 2; \qquad \text{and} \qquad s_2 \equiv *(b + 3) = 5;$$

$$
\begin{array}{lll}
\text{code}\ (s_1 s_2)\ \rho & = & \text{loadc } 7 \\
 & & \text{loadc } 2 \\
 & & \text{loadc } 10 \quad // \quad \textit{size of } \textbf{int}[10] \\
 & & \text{mul} \quad // \quad \textit{scaling} \\
 & & \text{add} \\
 & & \text{loadc } 17 \\
 & & \text{store} \\
 & & \text{pop} \quad // \quad \textit{end of } s_1 \\
 & & \\
 & & \text{loadc } 5 \\
 & & \text{loadc } 17 \\
 & & \text{load} \\
 & & \text{loadc } 3 \\
 & & \text{loadc } 1 \quad // \quad \textit{size of } \textbf{int} \\
 & & \text{mul} \quad // \quad \textit{scaling} \\
 & & \text{add} \\
 & & \text{store} \\
 & & \text{pop} \quad // \quad \textit{end of } s_2
\end{array}
$$

8 Freeing Occupied Storage

Problems:

- The freed storage area is still referenced by other pointers (dangling references).
- After several deallocations, the storage could look like this (fragmentation):



frei

Potential Solutions:

- Trust the programmer. Manage freed storage in a particular data structure (free list) $\Longrightarrow$ **malloc** or **free** my become expensive.
- Do nothing, i.e.:

$$\text{code } \textbf{free}\,(e);\ \rho \quad = \quad \begin{array}{l}\text{code}_{\text{R}}\ e\ \rho \\ \text{pop}\end{array}$$

  $\Longrightarrow$ simple and (in general) efficient.
- Use an automatic, potentially "conservative" Garbage-Collection, which occasionally collects certainly inaccessible heap space.

9 Functions

The definition of a function consists of

- a name, by which it can be called,
- a specification of the formal parameters;
- maybe a result type;
- a statement part, the body.

For C holds:

$$\text{code}_R\ f\ \rho \quad = \quad \text{loadc}\ \_f \quad = \quad \text{starting address of the code for } f$$

$\Longrightarrow$ Function names must also be managed in the address environment!

Example:

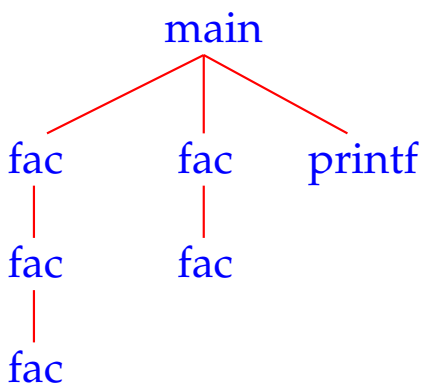
```
int fac (int x) {
    if (x ≤ 0) return 1;
    else return x * fac(x − 1);
}
```

```
main () {
    int n;
    n = fac(2) + fac(1);
    printf ("%d", n);
}
```

At any time during the execution, several instances of one function may exist, i.e., may have started, but not finished execution.

An instance is created by a call to the function.

The recursion tree in the example:



We conclude:

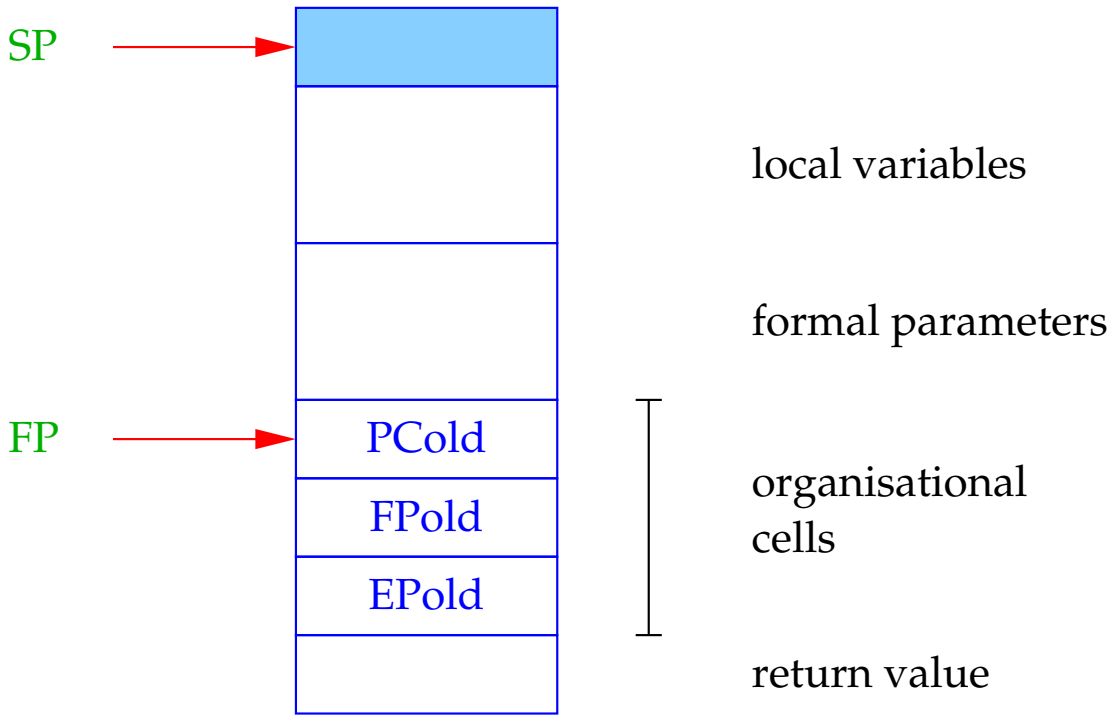
The formal parameters and local variables of the different instances of the same function must be kept separate.

Idea:

Allocate a special storage area for each instance of a function.

In sequential programming languages these storage areas can be managed on a stack. They are therefore called Stack Frames.

9.1 Storage Organization for Functions



FP $\widehat{=}$ Frame Pointer; points to the last organizational cell and is used to address the formal parameters and the local variables.

The caller must be able to continue execution in its frame after the return from a function. Therefore, at a function call the following values have to be saved into organizational cells:

- the FP
- the continuation address after the call and
- the actual EP.

Simplification: The return value fits into one storage cell.

Translation tasks for functions:

- Generate code for the body!
- Generate code for calls!

9.2 Computing the Address Environment

We have to distinguish two different kinds of variables:

1. globals, which are defined externally to the functions;
2. locals/automatic (including formal parameters), which are defined internally to the functions.

$$\Longrightarrow$$

The address environment ρ associates pairs $(tag, a) \in \{G, L\} \times \mathbb{N}_0$ with their names.

Note:

- There exist more refined notions of visibility of (the defining occurrences of) variables, namely nested blocks.
- The translation of different program parts in general uses different address environments!

Example (1):

```
0   int i;
    struct list {
      int info;
      struct list * next;
    } * l;

1   int ith (struct list * x, int i) {
      if (i ≤ 1) return x →info;
      else return ith (x →next, i − 1);
    }

2   main () {
      int k;
      scanf ("%d", &i);
      scanlist (&l);
      printf ("\n\t%d\n", ith (l,i));
    }
```

address	environment		at 0
ρ_0 :	i	$\mapsto$	$(G, 1)$
	l	$\mapsto$	$(G, 2)$
	ith	$\mapsto$	$(G, \_\text{ith})$
	main	$\mapsto$	$(G, \_\text{main})$
		$\mapsto$	...

Example (2):

```
0   int i;
    struct list {
      int info;
      struct list * next;
    } * l;

1   int ith (struct list * x, int i) {
      if (i ≤ 1) return x →info;
      else return ith (x →next, i − 1);
    }

2   main () {
      int k;
      scanf ("%d", &i);
      scanlist (&l);
      printf ("\n\t%d\n", ith (l,i));
    }
```

---

1 inside of ith:

ρ_1 :

i	$\mapsto$	$(L, 2)$
x	$\mapsto$	$(L, 1)$
l	$\mapsto$	$(G, 2)$
ith	$\mapsto$	$(G, \_ith)$
main	$\mapsto$	$(G, \_main)$
	$\mapsto$	$\ldots$

Example (3):

```
0   int i;
    struct list {
      int info;
      struct list * next;
    } * l;

1   int ith (struct list * x, int i) {
      if (i ≤ 1) return x →info;
      else return ith (x →next, i − 1);
    }

2   main () {
      int k;
      scanf ("%d", &i);
      scanlist (&l);
      printf ("\n\t%d\n", ith (l,i));
    }
```

---

2 inside of main:

ρ_2 :

i	$\mapsto$	$(G, 1)$
l	$\mapsto$	$(G, 2)$
k	$\mapsto$	$(L, 1)$
ith	$\mapsto$	$(G, \_ith)$
main	$\mapsto$	$(G, \_main)$
	$\ldots$	

9.3 Calling/Entering and Leaving Functions

Be f the actual function, the Caller, and let f call the function g, the Callee.

The code for a function call has to be distributed among the Caller and the Callee:

The distribution depends on who has which information.

Actions upon calling/entering g:

1. Saving FP, EP — mark
2. Computing the actual parameters
3. Determining the start address of g
4. Setting the new FP — call
5. Saving PC and jump to the beginning of g — call
6. Setting the new EP — enter
7. Allocating the local variables — alloc

(mark, call, enter, alloc: available in f)

(enter, alloc: available in g)

Actions upon leaving g:

1. Restoring the registers FP, EP, SP — return
2. Returning to the code of f, i.e. restoring the PC — return

Altogether we generate for a call:

$$
\begin{array}{lcl}
\text{code}_R\ g(e_1, \ldots, e_n)\ \rho & = & \text{mark} \\
 & & \text{code}_R\ e_1\ \rho \\
 & & \ldots \\
 & & \text{code}_R\ e_m\ \rho \\
 & & \text{code}_R\ g\ \rho \\
 & & \text{call n}
\end{array}
$$

where n = space for the actual parameters

Note:

- Expressions occurring as actual parameters will be evaluated to their R-value $\Longrightarrow$ Call-by-Value-parameter passing.
- Function g can also be an expression, whose R-value is the start address of the function to be called ...

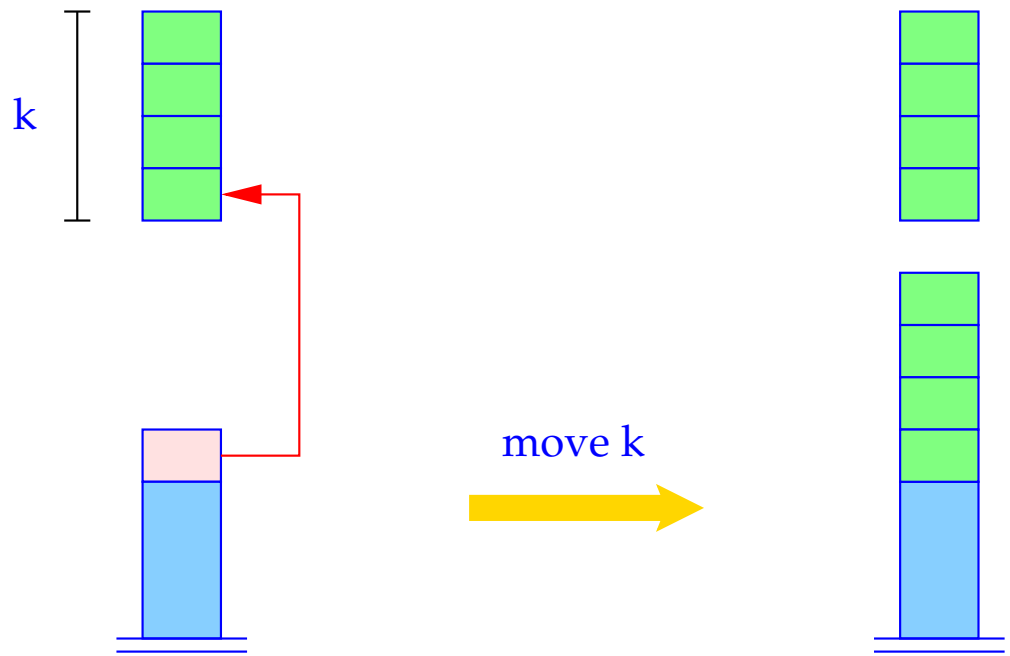
- Function names are regarded as constant pointers to functions, similarly to declared arrays. The R-value of such a pointer is the start address of the function.
- For a variable **int** $(*)()\ g;$, the two calls

  $$(*g)() \qquad \text{und} \qquad g()$$

  are equivalent :-)
  Normalization: Dereferencing of a function pointer is ignored.
- Structures are copied when they are passed as parameters.

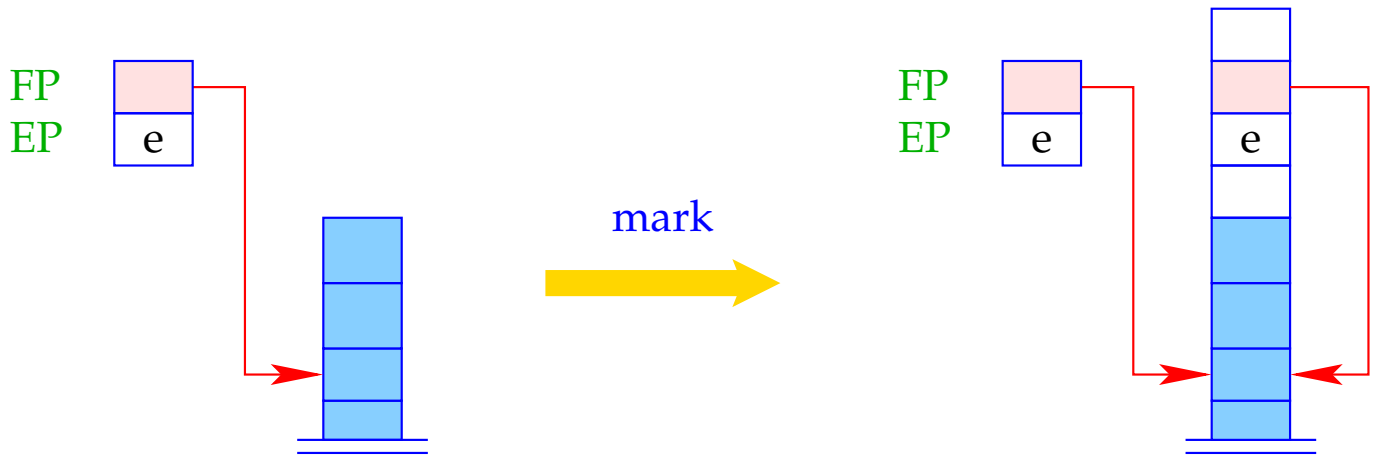
In consequence:

$\text{code}_R\ f\ \rho$	$=$	$\text{loadc}\ (\rho\, f)$	f a function name
$\text{code}_R\ (*e)\ \rho$	$=$	$\text{code}_R\ e\ \rho$	e a function pointer
$\text{code}_R\ e\ \rho$	$=$	$\text{code}_L\ e\ \rho$	
		move k	e a structure of size k




```
for (i = k-1; i≥0; i--)
        S[SP+i] = S[S[SP]+i];
SP = SP+k–1;
```

The instruction mark allocates space for the return value and for the organizational cells and saves the FP and EP.

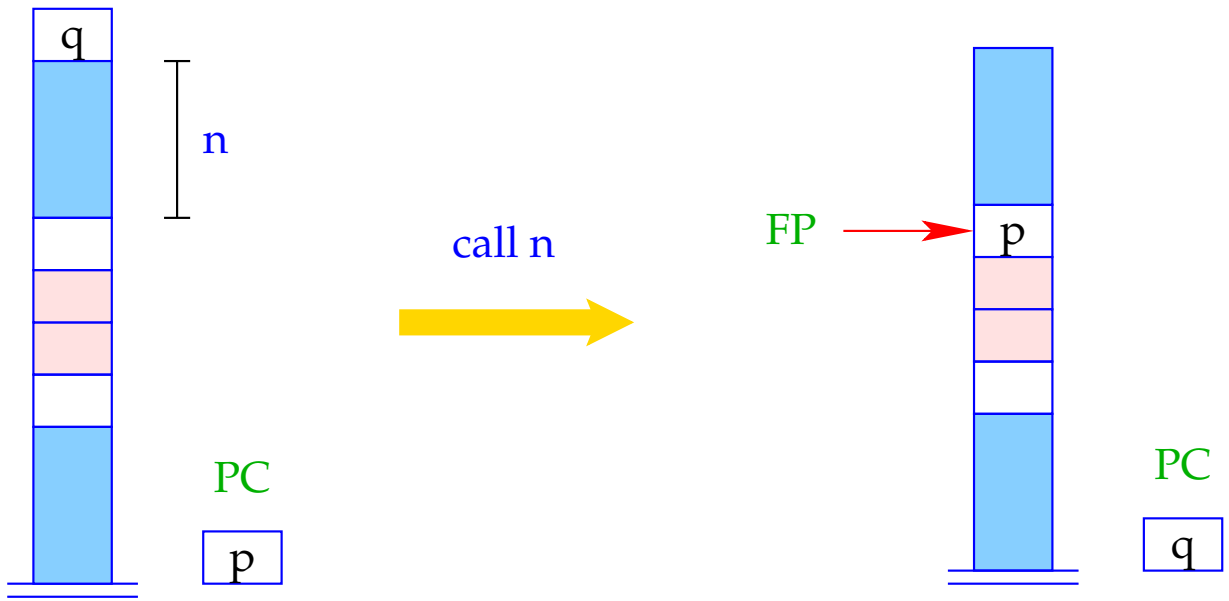


S[SP+2] = EP;

S[SP+3] = FP;

SP = SP + 4;

The instruction call n saves the continuation address and assigns FP, SP, and PC their new values.




```
FP = SP - n - 1;
S[FP] = PC;
PC = S[SP];
SP--;
```

Correspondingly, we translate a function definition:

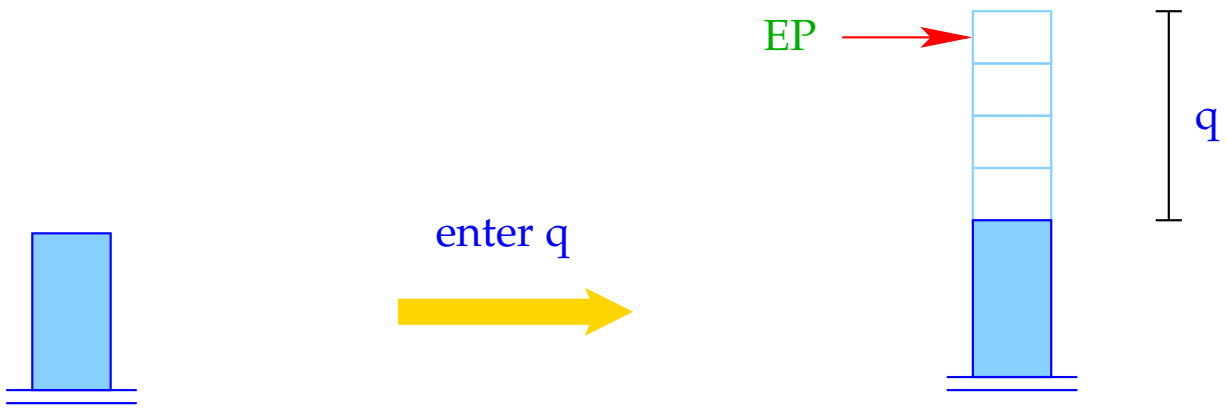
$$\text{code}\quad t\ \mathrm{f}\ (specs)\{V\_defs\quad ss\}\quad \rho\quad =$$

$$\begin{array}{lll} \mathrm{\_f:} & \text{enter q} & // \quad \textit{Setting the EP} \\ & \text{alloc k} & // \quad \textit{Allocating the local variables} \\ & \text{code } ss\ \rho_{\mathrm{f}} & \\ & \text{return} & // \quad \textit{leaving the function} \end{array}$$

where

t	=	return type of f with $\lvert t \rvert \leq 1$	
q	=	$maxS + \mathrm{k}$	where
$maxS$	=	maximal depth of the local stack	
k	=	space for the local variables	
ρ_{f}	=	address environment for f	
		// *takes care of specs, V_defs* and ρ	

The instruction enter q sets EP to its new value. Program execution is terminated if not enough space is available.

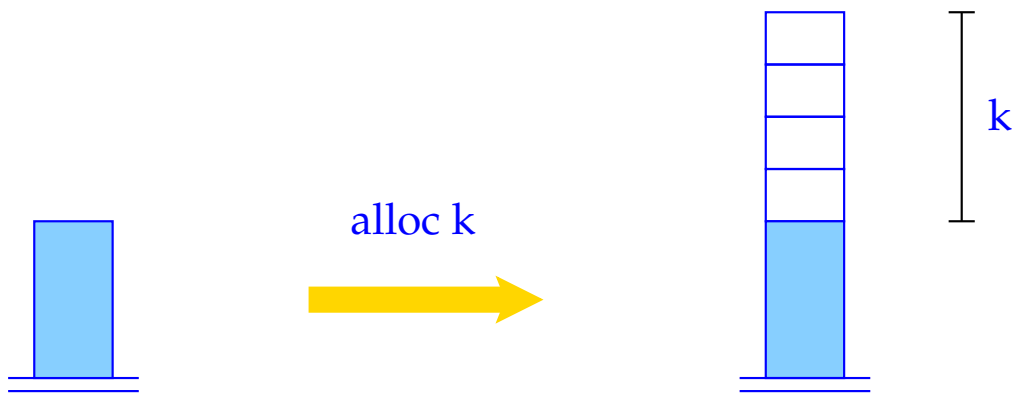


EP = SP + q;

if (EP $\geq$ NP)

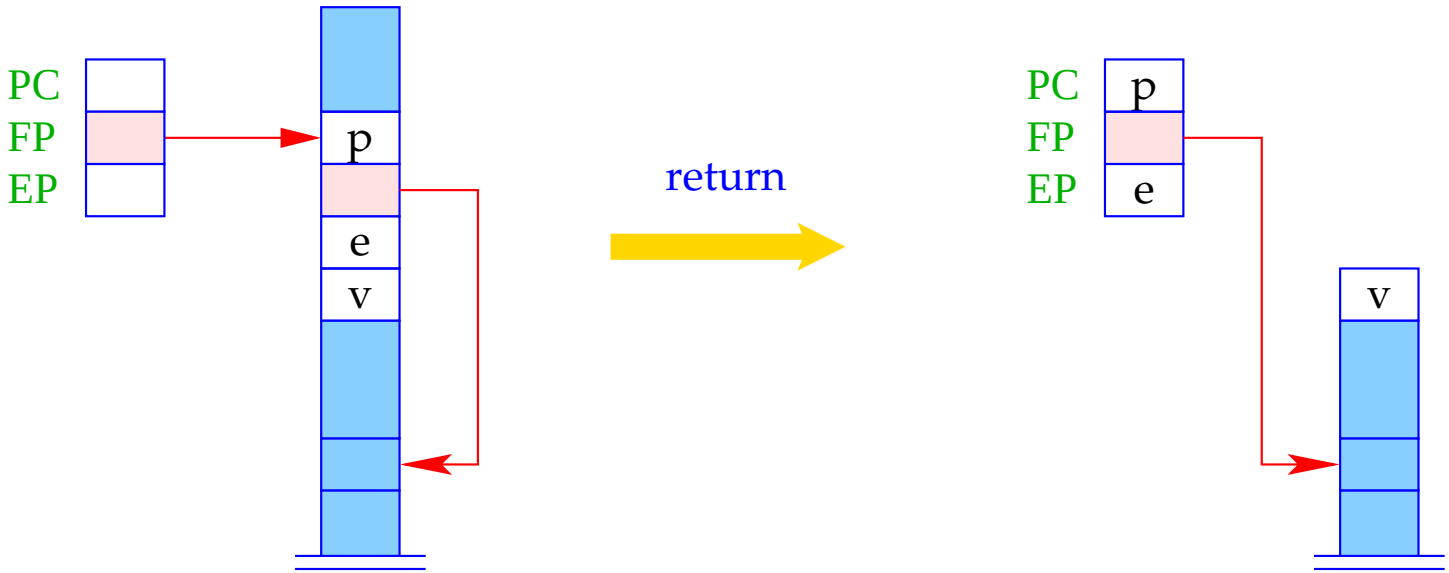
Error ("Stack Overflow");

The instruction alloc k reserves stack space for the local variables.



SP = SP + k;

The instruction return pops the actual stack frame, i.e., it restores the registers PC, EP, SP, and FP and leaves the return value on top of the stack.



PC = S[FP]; EP = S[FP-2];

if (EP $\geq$ NP) Error ("Stack Overflow");

SP = FP-3; FP = S[SP+2];

9.4 Access to Variables and Formal Parameters, and Return of Values

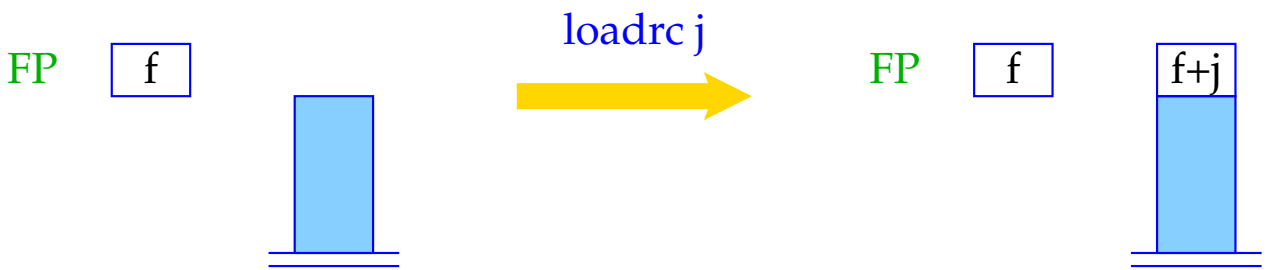
Local variables and formal parameters are addressed relative to the current FP.

We therefore modify code_L for the case of variable names.

For $\rho\ x = (tag, j)$ we define

$$\text{code}_\text{L}\ x\ \rho = \begin{cases} \text{loadc j} & tag = G \\ \text{loadrc j} & tag = L \end{cases}$$

The instruction loadrc j computes the sum of FP and j.



SP++;

S[SP] = FP+j;

As an optimization one introduces the instructions loadr j and storer j . This is analogous to loada j and storea j.

$$\begin{array}{lcl} \text{loadr j} & = & \text{loadrc j} \\ & & \text{load} \end{array}$$

$$\begin{array}{lcl} \text{storer j} & = & \text{loadrc j} \\ & & \text{store} \end{array}$$

The code for **return** *e;* corresponds to an assignment to a variable with relative address -3.

$$\begin{array}{lcl} \text{code } \textbf{return}\, e;\ \rho & = & \text{code}_R\ e\ \rho \\ & & \text{storer -3} \\ & & \text{return} \end{array}$$

Example: For the function

```
int fac (int x) {
    if (x ≤ 0) return 1;
    else return x * fac (x − 1);
}
```

we generate:

```
_fac:  enter q          loadc 1          A:  loadr 1          mul
       alloc 0          storer -3            mark            storer -3
       loadr 1          return               loadr 1         return
       loadc 0          jump B               loadc 1    B:   return
       leq                                   sub
       jumpz A                               loadc _fac
                                             call 1
```

where $\rho_{\text{fac}} : x \mapsto (L, 1)$ and $q = 1 + 4 + 2 = 7$.

10 Translation of Whole Programs

The state before program execution starts:

$$\text{SP} = -1 \qquad \text{FP} = \text{EP} = 0 \qquad \text{PC} = 0 \qquad \text{NP} = \text{MAX}$$

Be $p \equiv V\_defs \quad F\_def_1 \ \ldots \ F\_def_n$, a program, where F_def_i defines a function f_i, of which one is named main.

The code for the program p consists of:

- Code for the function definitions F_def_i;
- Code for allocating the global variables;
- Code for the call of main();
- the instruction halt.

We thus define:

$$
\begin{array}{lll}
\text{code } p\ \emptyset & = & \text{enter } (\mathrm{k}+6) \\
 & & \text{alloc } (\mathrm{k}+1) \\
 & & \text{mark} \\
 & & \text{loadc \_main} \\
 & & \text{call } 0 \\
 & & \text{pop} \\
 & & \text{halt} \\
 & \_\mathrm{f}_1\text{:} & \text{code } F\_def_1\ \rho \\
 & & \vdots \\
 & \_\mathrm{f}_n\text{:} & \text{code } F\_def_n\ \rho
\end{array}
$$

where

$$
\begin{array}{lcl}
\emptyset & \widehat{=} & \text{empty address environment;} \\
\rho & \widehat{=} & \text{global address environment;} \\
\mathrm{k} & \widehat{=} & \text{space for global variables} \\
\text{\_main} & \in & \{\_\mathrm{f}_1, \ldots, \_\mathrm{f}_n\}
\end{array}
$$

The Translation of Functional Programming Languages

11 The language PuF

We only regard a mini-language PuF ("Pure Functions").

We do not treat, as yet:

- Side effects;
- Data structures.

A Program is an expression e of the form:

$$
\begin{array}{rcl}
e & ::= & b \mid x \mid (\square_1\ e) \mid (e_1\ \square_2\ e_2) \\
& \mid & (\textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2) \\
& \mid & (e'\ e_0 \ldots e_{k-1}) \\
& \mid & (\textbf{fn}\ x_0, \ldots, x_{k-1} \Rightarrow e) \\
& \mid & (\textbf{let}\ x_1 = e_1; \ldots; x_n = e_n\ \textbf{in}\ e_0) \\
& \mid & (\textbf{letrec}\ x_1 = e_1; \ldots; x_n = e_n\ \textbf{in}\ e_0)
\end{array}
$$

An expression is therefore

- a basic value, a variable, the application of an operator, or
- a function-application, a function-abstraction, or
- a **let**-expression, i.e. an expression with locally defined variables, or
- a **letrec**-expression, i.e. an expression with simultaneously defined local variables.

For simplicity, we only allow `int` and `bool` as basic types.

Example:

The following well-known function computes the factorial of a natural number:

$$\begin{array}{lll} \textbf{letrec}\ \text{fac} & = & \textbf{fn}\ x \Rightarrow \textbf{if}\ x \leq 1\ \textbf{then}\ 1 \\ & & \textbf{else}\ x \cdot \text{fac}\ (x-1) \\ \textbf{in}\ \text{fac}\ 7 & & \end{array}$$

As usual, we only use the minimal amount of parentheses.

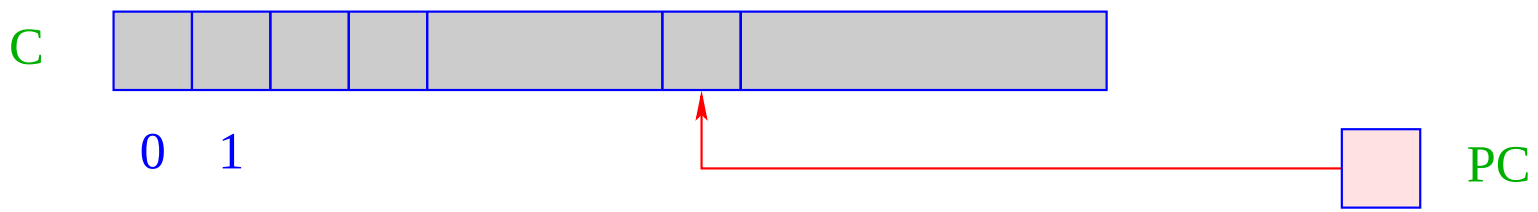
There are two Semantics:

CBV: Arguments are evaluated before they are passed to the function (as in SML);

CBN: Arguments are passed unevaluated; they are only evaluated when their value is needed (as in Haskell).

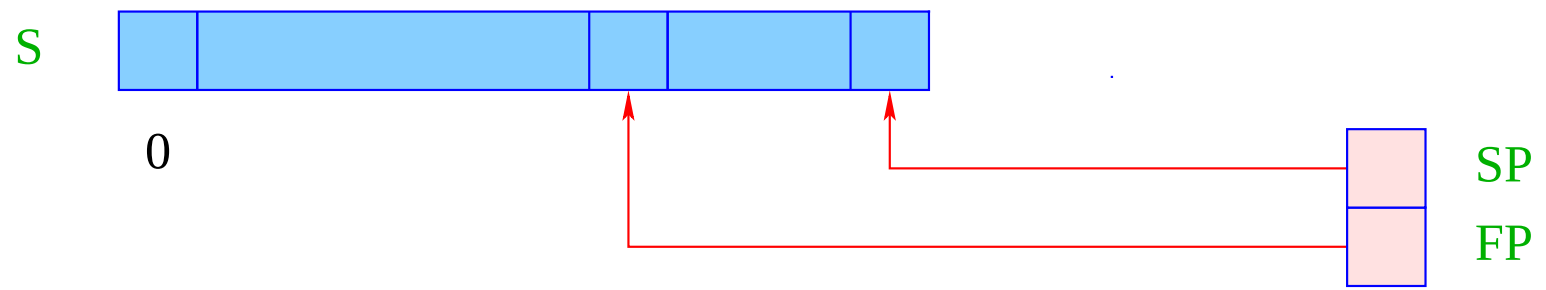
12 Architecture of the MaMa:

We know already the following components:



C = Code-store – contains the MaMa-program;
each cell contains one instruction;

PC = Program Counter – points to the instruction to be executed next;

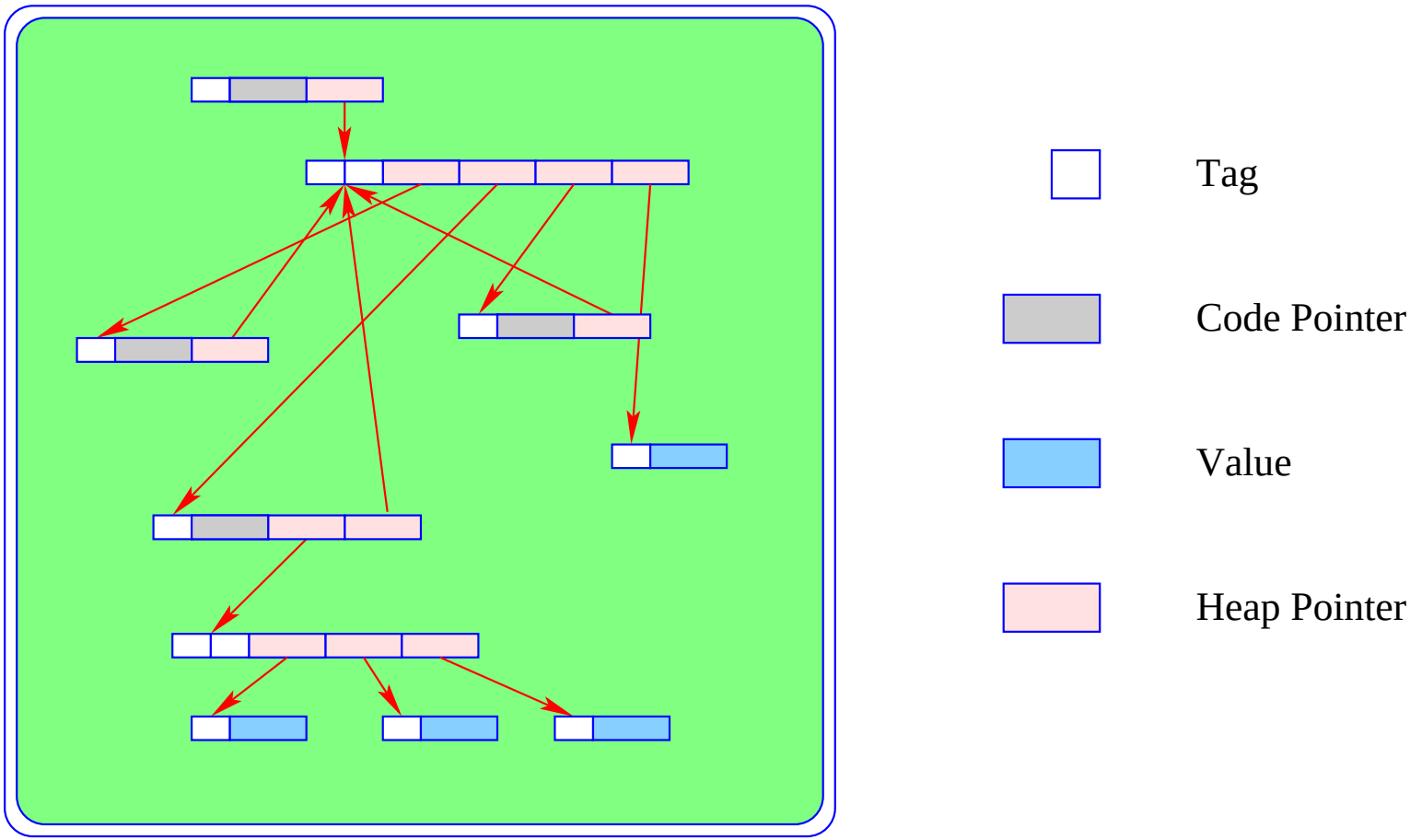


S = Runtime-Stack – each cell can hold a basic value or an address;

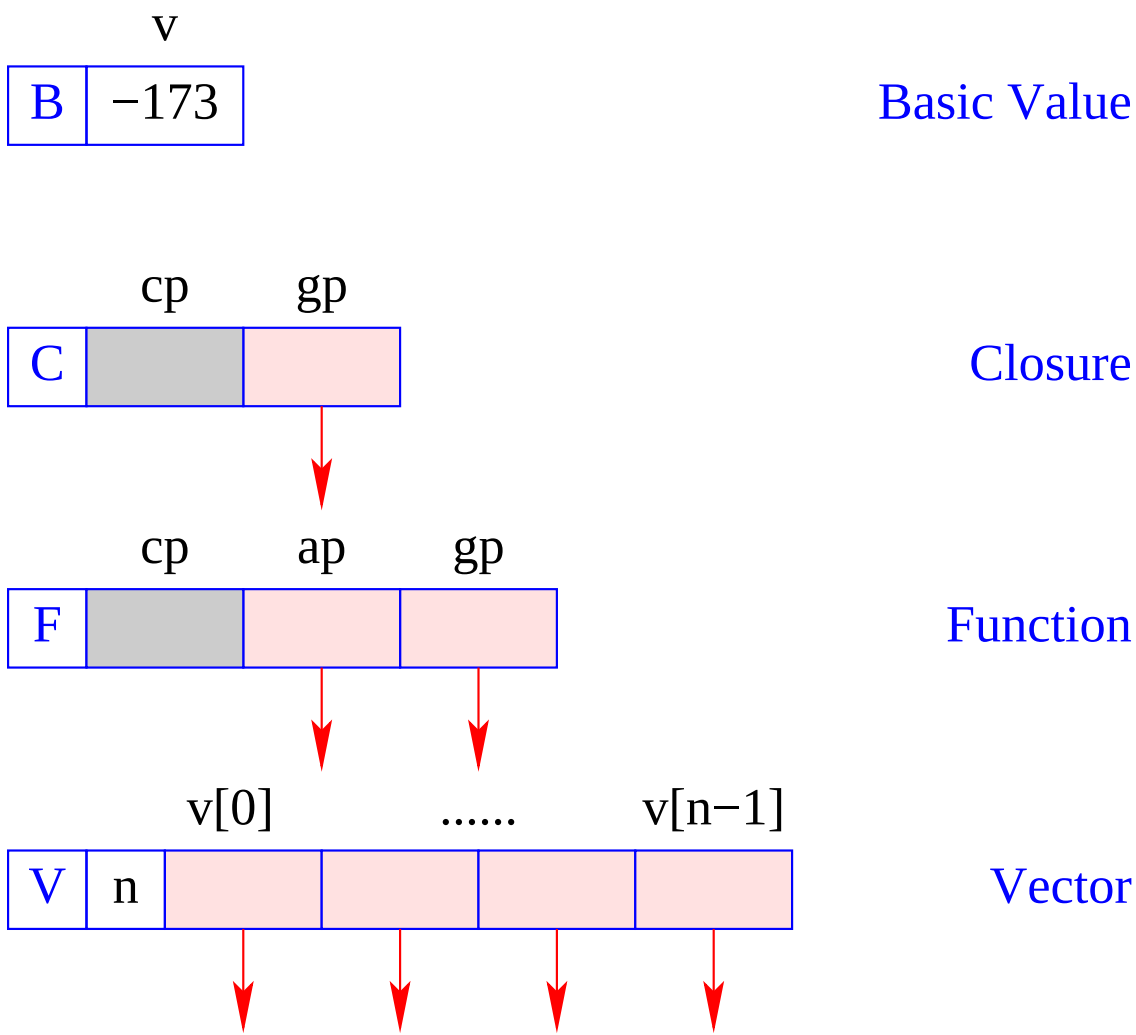
SP = Stack-Pointer – points to the topmost occupied cell;
as in the CMa implicitely represented;

FP = Frame-Pointer – points to the actual stack frame.

We also need a heap H:



... it can be thought of as an abstract data type, being capable of holding data objects of the following form:



The instruction new $(tag, args)$ creates a corresponding object (B, C, F, V) in H and returns a reference to it.

We distinguish three different kinds of code for an expression e:

- $\text{code}_V\ e$ — (generates code that) computes the Value of e, stores it in the heap and returns a reference to it on top of the stack (the normal case);
- $\text{code}_B\ e$ — computes the value of e, and returns it on the top of the stack (only for Basic types);
- $\text{code}_C\ e$ — does not evaluate e, but stores a Closure of e in the heap and returns a reference to the closure on top of the stack.

We start with the code schemata for the first two kinds:

13 Simple expressions

Expressions consisting only of constants, operator applications, and conditionals are translated like expressions in imperative languages:

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

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

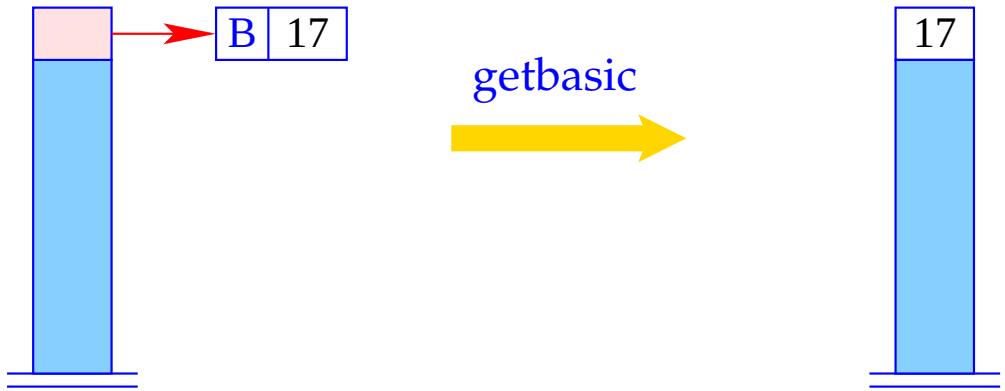
Note:

- ρ denotes the actual address environment, in which the expression is translated. Address environments have the form:

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

- The extra argument sd, the stack difference, *simulates* the movement of the SP when instruction execution modifies the stack. It is needed later to address variables.
- The instructions op_1 and op_2 implement the operators $\square_1$ and $\square_2$, in the same way as the the operators neg and add implement negation resp. addition in the CMa.
- For all other expressions, we first compute the value in the heap and then dereference the returned pointer:

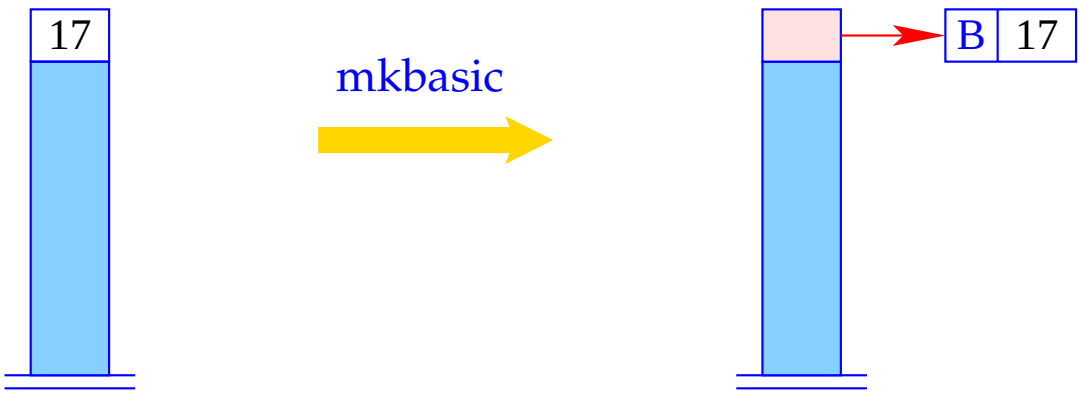
$$\begin{aligned} \text{code}_B\ e\ \rho\ \text{sd} \quad &= \quad \text{code}_V\ e\ \rho\ \text{sd} \\ & \qquad\quad \text{getbasic} \end{aligned}$$




```
if (H[S[SP]] != (B,_))
      Error “not basic!”;
else
      S[SP] = H[S[SP]].v;
```

For code_V and simple expressions, we define analogously:

$$
\begin{array}{lcll}
\text{code}_V\ b\ \rho\ \text{sd} & = & & \text{loadc b; mkbasic} \\
\text{code}_V\ (\square_1\ e)\ \rho\ \text{sd} & = & & \text{code}_B\ e\ \rho\ \text{sd} \\
 & & & \text{op}_1\text{; mkbasic} \\
\text{code}_V\ (e_1\ \square_2\ e_2)\ \rho\ \text{sd} & = & & \text{code}_B\ e_1\ \rho\ \text{sd} \\
 & & & \text{code}_B\ e_2\ \rho\ (\text{sd}+1) \\
 & & & \text{op}_2\text{; mkbasic} \\
\text{code}_V\ (\textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2)\ \rho\ \text{sd} & = & & \text{code}_B\ e_0\ \rho\ \text{sd} \\
 & & & \text{jumpz A} \\
 & & & \text{code}_V\ e_1\ \rho\ \text{sd} \\
 & & & \text{jump B} \\
 & & \text{A:} & \text{code}_V\ e_2\ \rho\ \text{sd} \\
 & & \text{B:} & \ldots
\end{array}
$$



S[SP] = new (B,S[SP]);

14 Accessing Variables

We must distinguish between local and global variables.

Example: Regard the function f :

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

The function f uses the global variable c and the local variables a (as formal parameter) and b (introduced by the inner **let**).

The binding of a global variable is determined, when the function is constructed (static scoping!), and later only looked up.

Accessing Global Variables

- The bindings of global variables of an expression or a function are kept in a vector in the heap (Global Vector).
- They are addressed consecutively starting with 0.
- When an F-object or a C-object are constructed, the Global Vector for the function or the expression is determined and a reference to it is stored in the gp-component of the object.
- During the evaluation of an expression, the (new) register GP (Global Pointer) points to the actual Global Vector.
- In constrast, local variables should be administered on the stack ...

$\Longrightarrow$ General form of the address environment:

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

Accessing Local Variables

Local variables are administered on the stack, in stack frames.

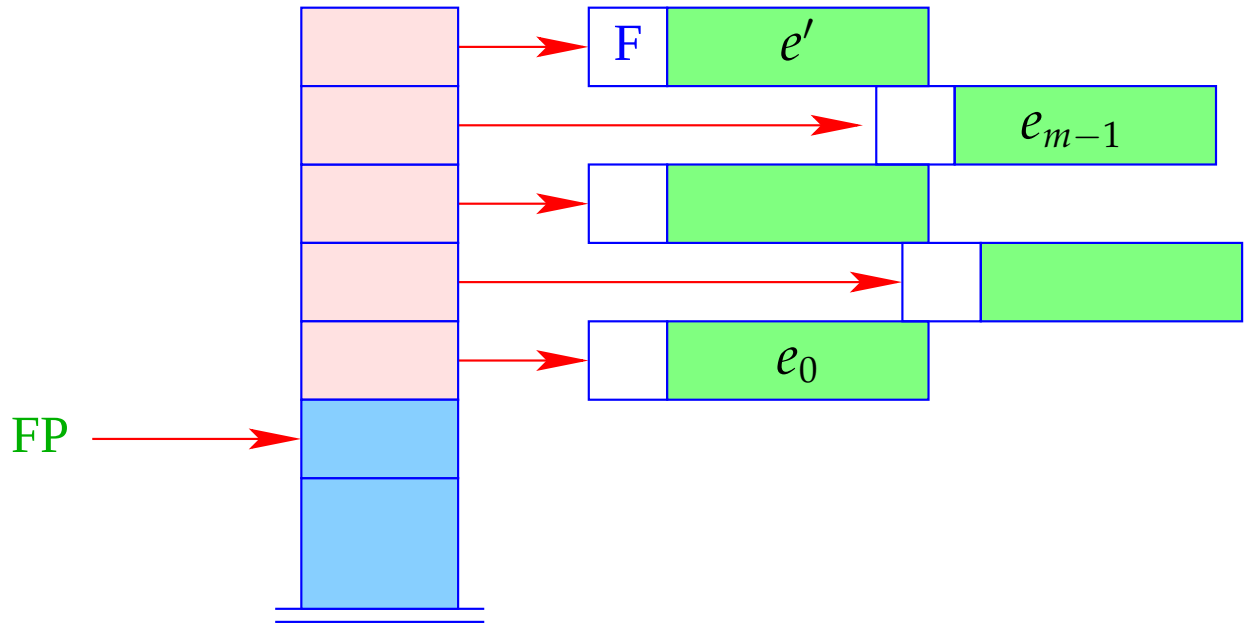
Let $e \equiv e'\ e_0\ \ldots\ e_{m-1}$ be the application of a function e' to arguments $e_0, \ldots, e_{m-1}$.

Warning:

The arity of e' does not need to be m :-)

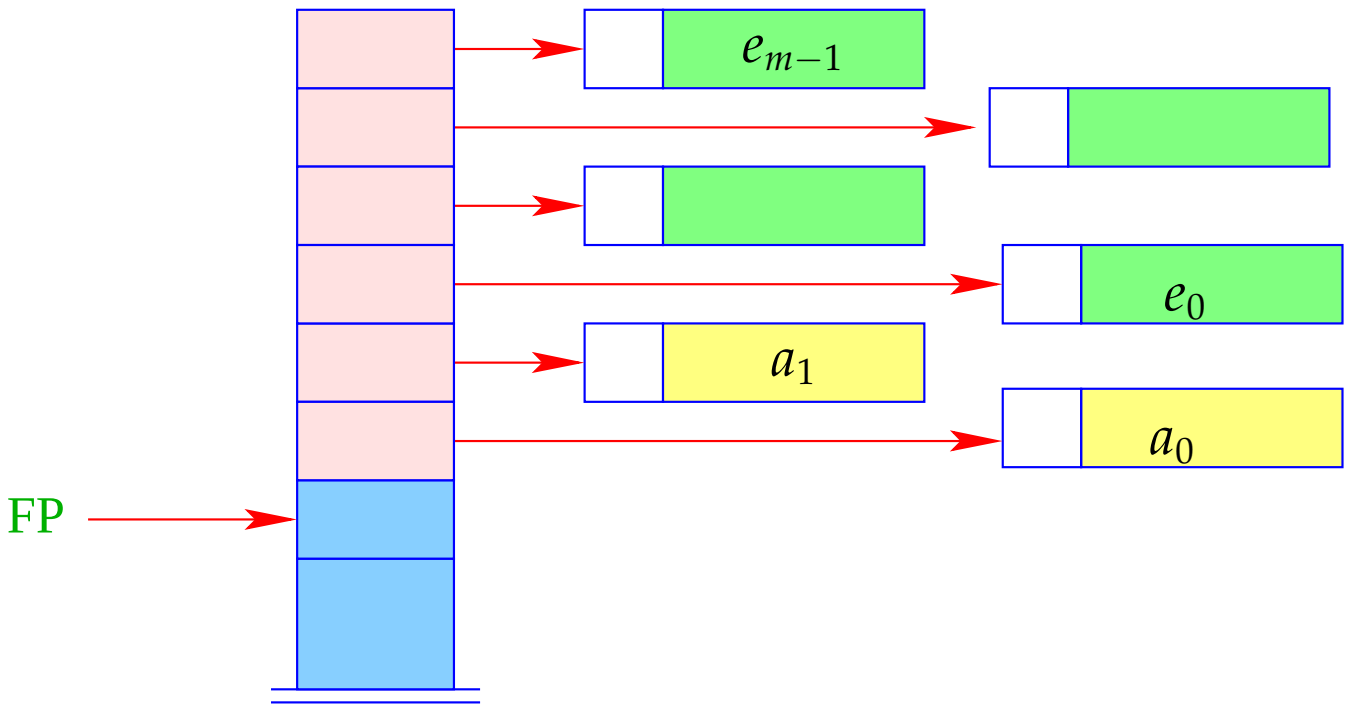
- PuF functions have curried types, $f : t_1 \to t_2 \to \ldots \to t_n \to t$
- f may therefore receive less than n arguments (under supply);
- f may also receive more than n arguments, if t is a functional type (over supply).

Possible stack organisations:

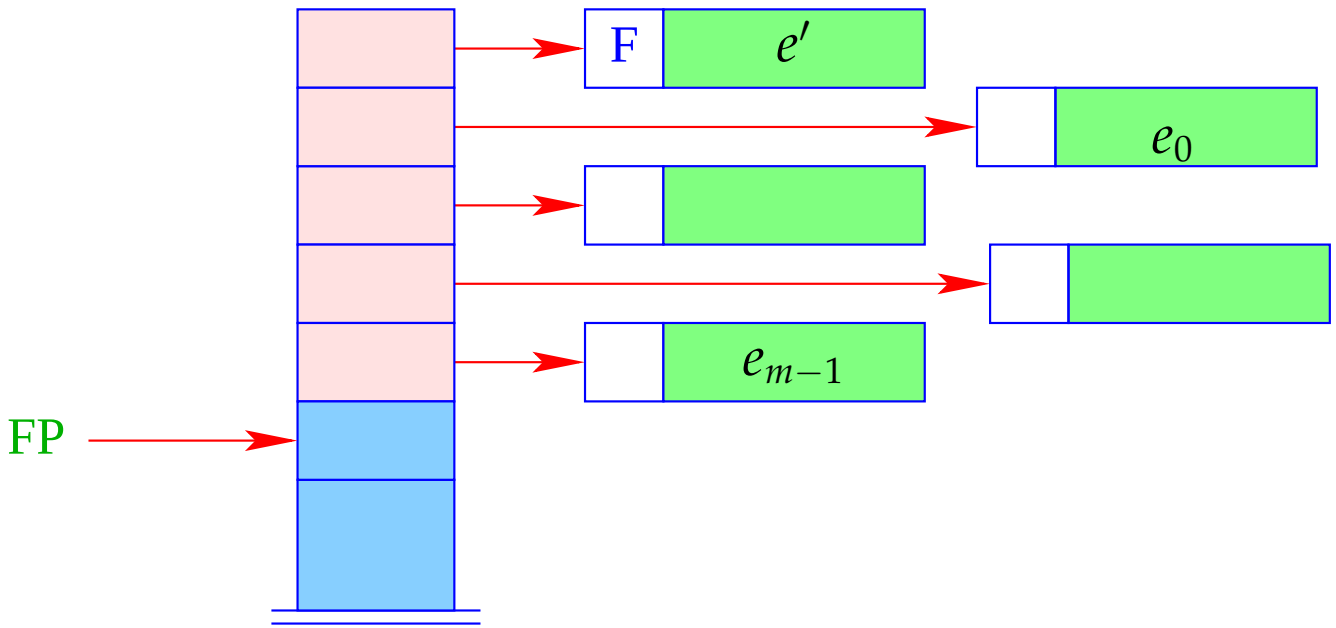


+ Addressing of the arguments can be done relative to FP
– The local variables of e' cannot be addressed relative to FP.
– If e' is an n-ary function with $n < m$, i.e., we have an over-supplied function application, the remaining $m - n$ arguments will have to be shifted.

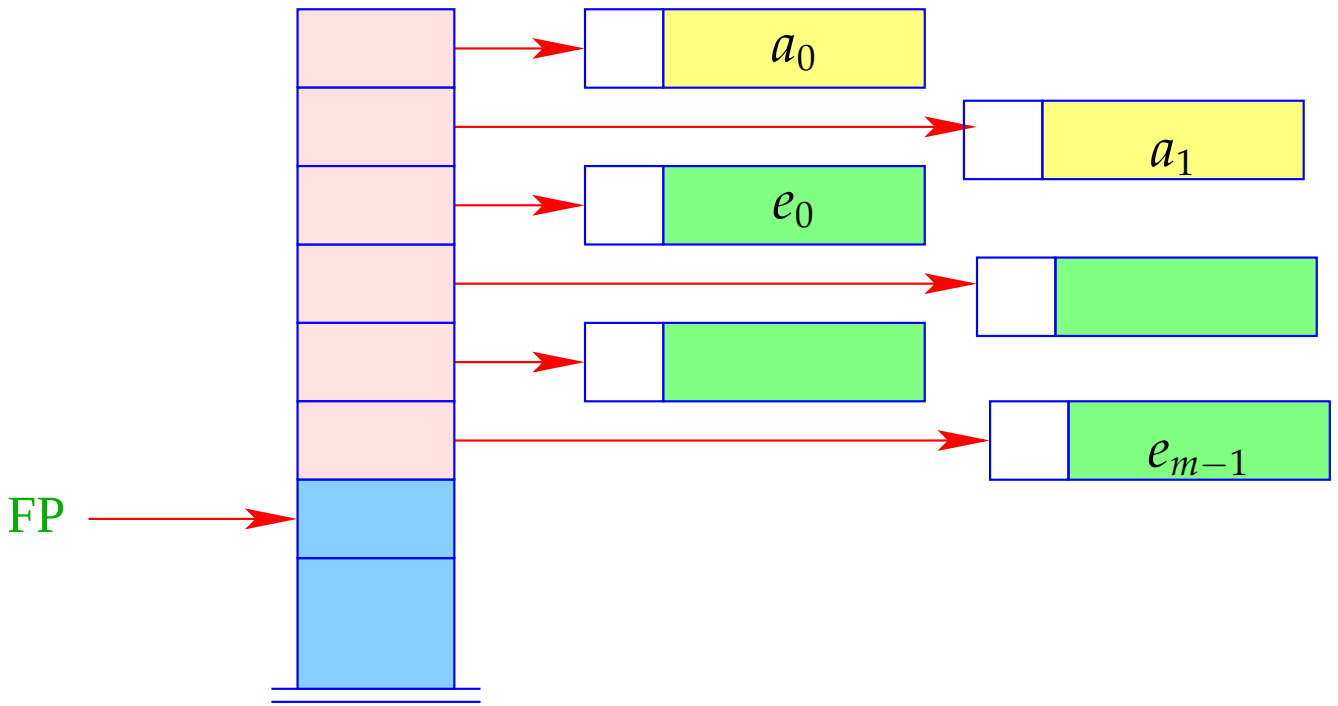
– If e' evaluates to a function, which has already been partially applied to the parameters $a_0, \ldots, a_{k-1}$, these have to be sneaked in underneath e_0:



Alternative:



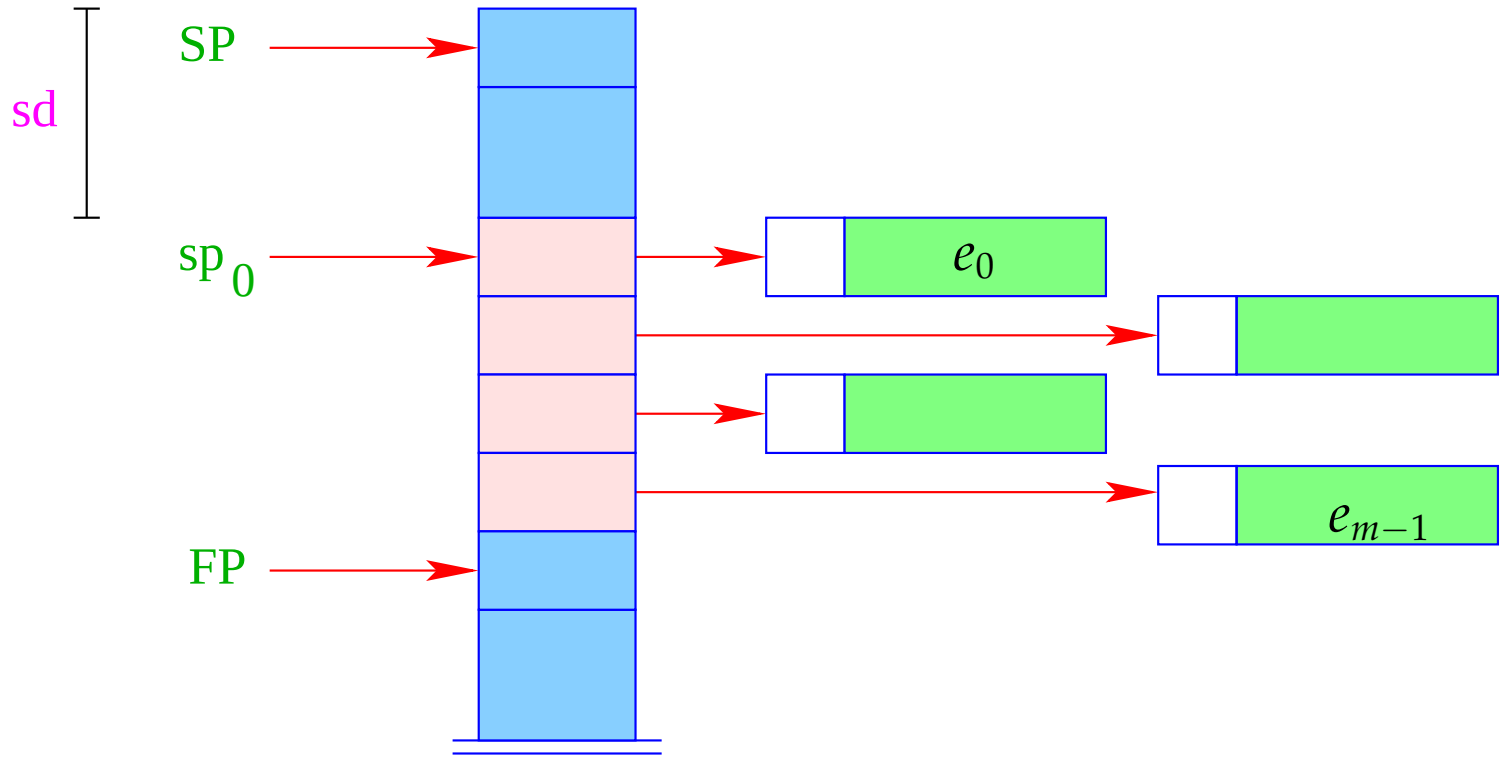
+ The further arguments $a_0, \ldots, a_{k-1}$ and the local variables can be allocated above the arguments.



– 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:

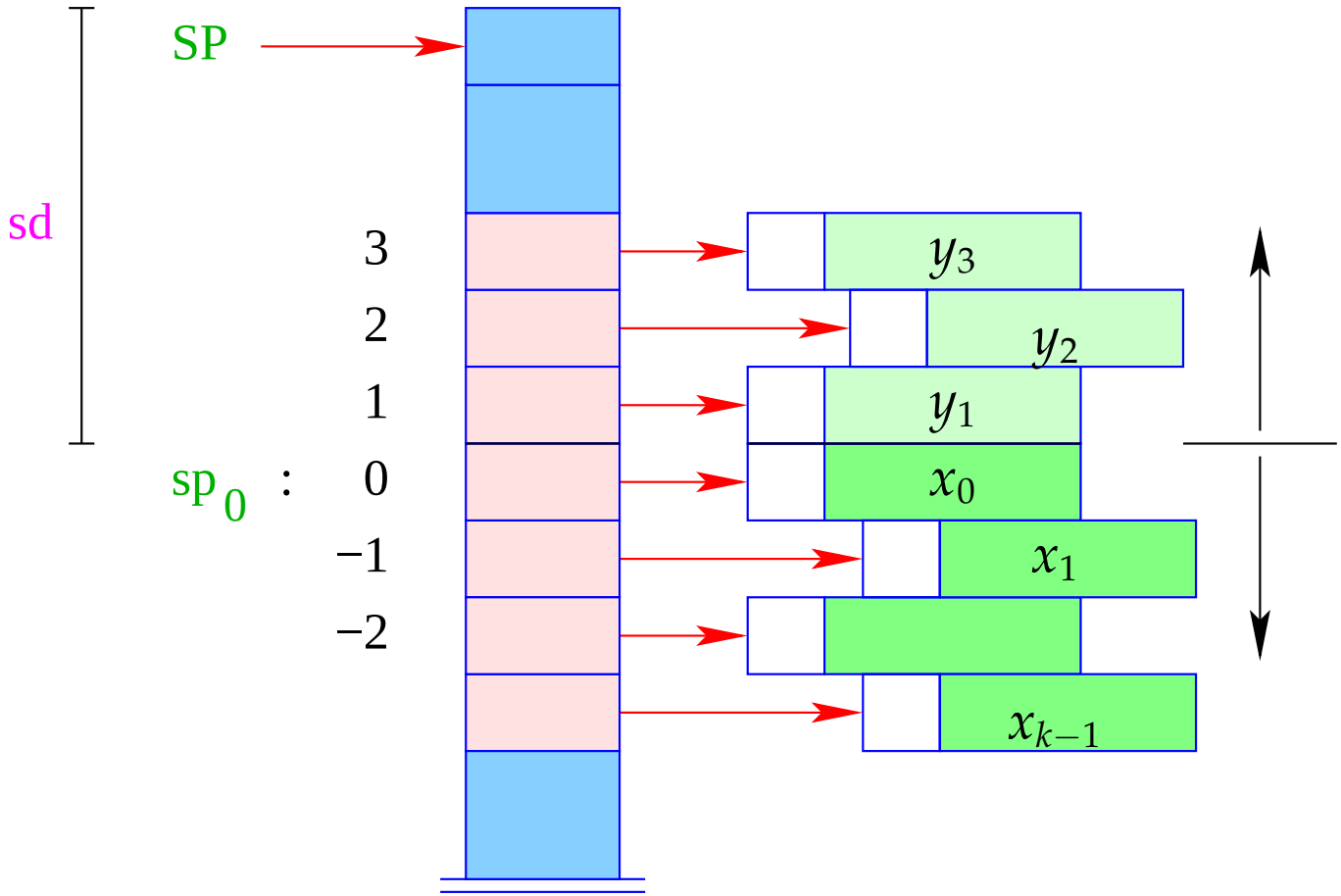
- We address both, arguments and local variables, relative to the stack pointer SP !!!
- However, the stack pointer changes during program execution...



- The differerence between the current value of SP and its value 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 $\mathrm{sp}_0 + i = (\mathrm{SP} - \mathrm{sd}) + i$

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

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

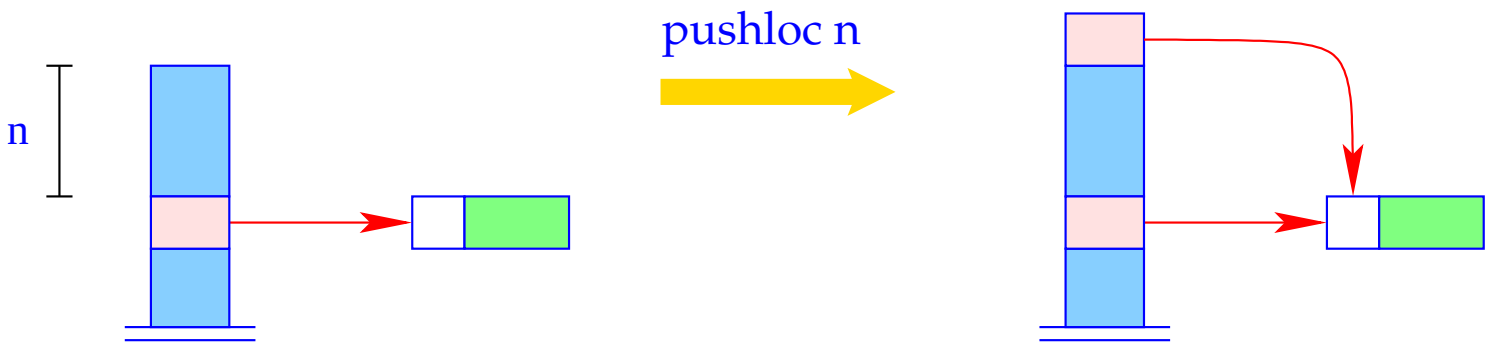
The instruction eval checks, whether the value has already been computed or whether its evaluation has to yet to be done ($\Longrightarrow$ will be treated later :-)

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++;

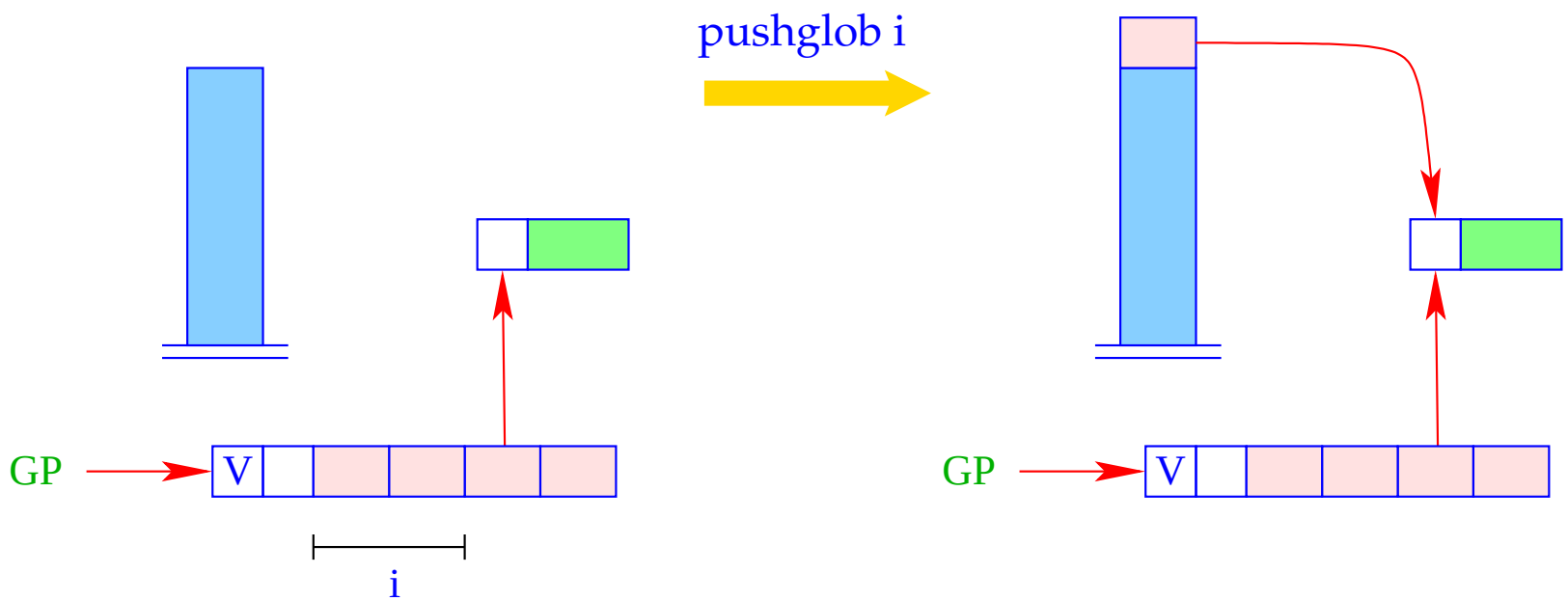
Correctness argument:

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 $\mathrm{sd} = 1$.

With CBN, we obtain:

$\mathrm{code}_V\ e\ \rho\ 1$	$=$	getvar b ρ 1	$=$	1	pushloc 0
		eval		2	eval
		getbasic		2	getbasic
		getvar c ρ 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:

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

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

In the case of CBV, we use 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:

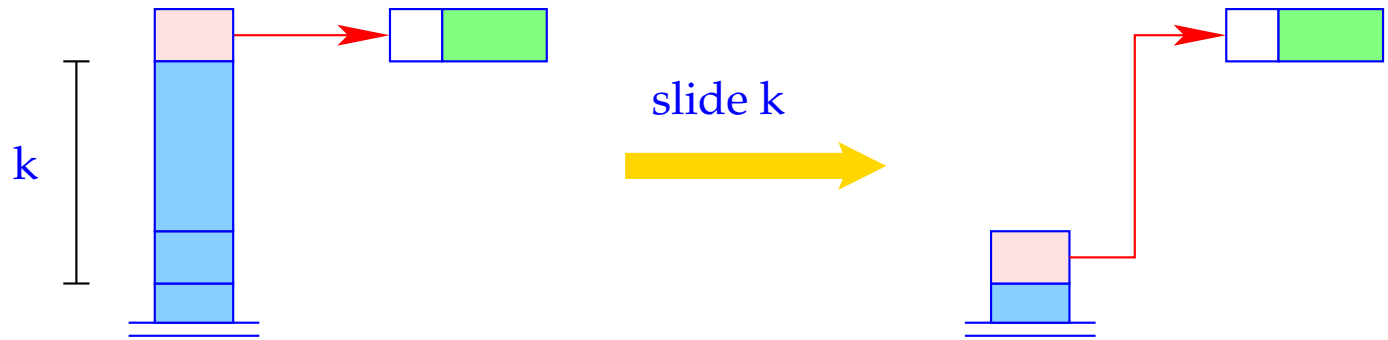
Consider the expression

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

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 these vectors and the start address of the code for the body;

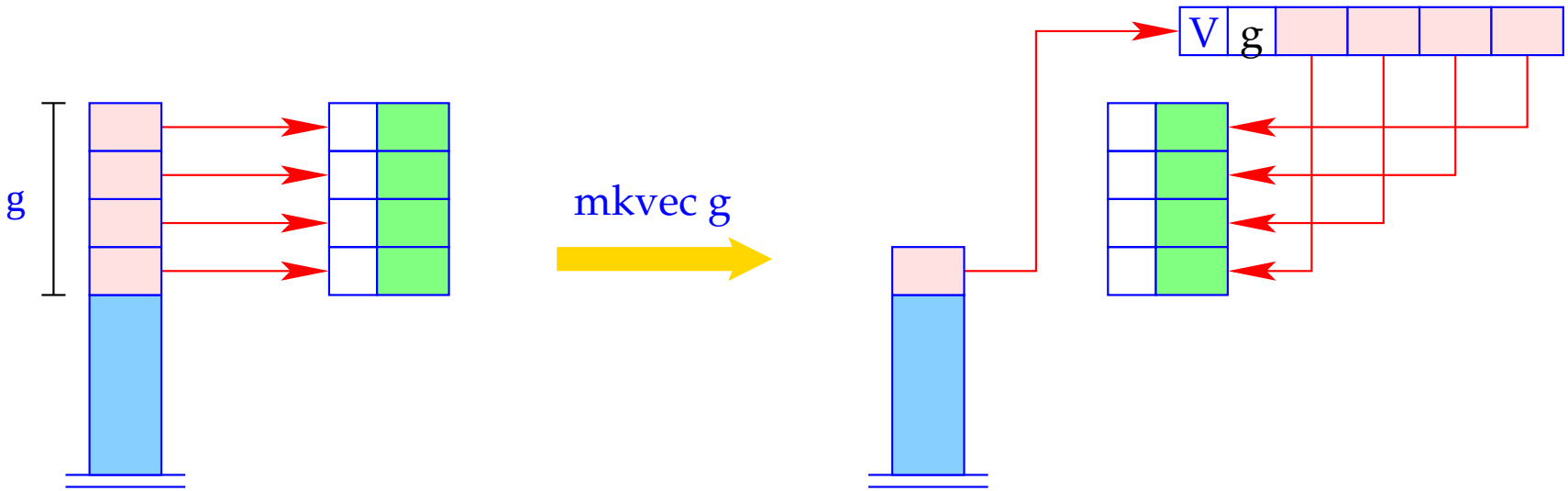
Separately, code for the body has to be generated.

Thus:

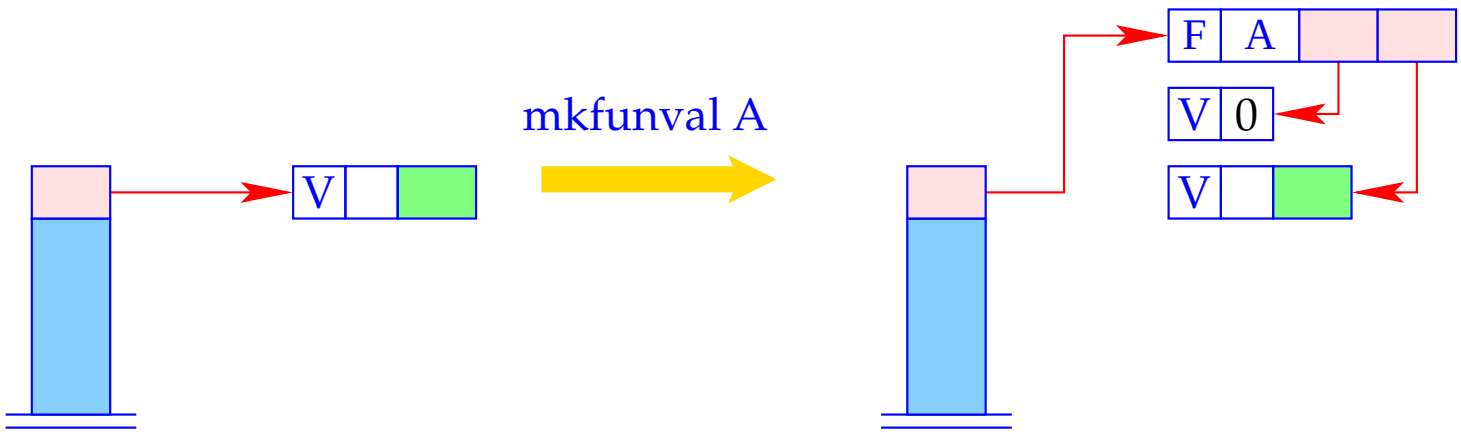
$$
\begin{array}{lll}
\text{code}_V\ (\textbf{fn}\ x_0, \ldots, x_{k-1} \Rightarrow e)\ \rho\ \text{sd} & = & \text{getvar}\ z_0\ \rho\ \text{sd} \\
 & & \text{getvar}\ z_1\ \rho\ (\text{sd}+1) \\
 & & \ldots \\
 & & \text{getvar}\ z_{g-1}\ \rho\ (\text{sd}+g-1) \\
 & & \text{mkvec g} \\
 & & \text{mkfunval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

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

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

$\mathrm{code}_V\ f\ \rho\ 1$ 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:

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

To implement CBV, we use code_V instead of 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:  ...
```

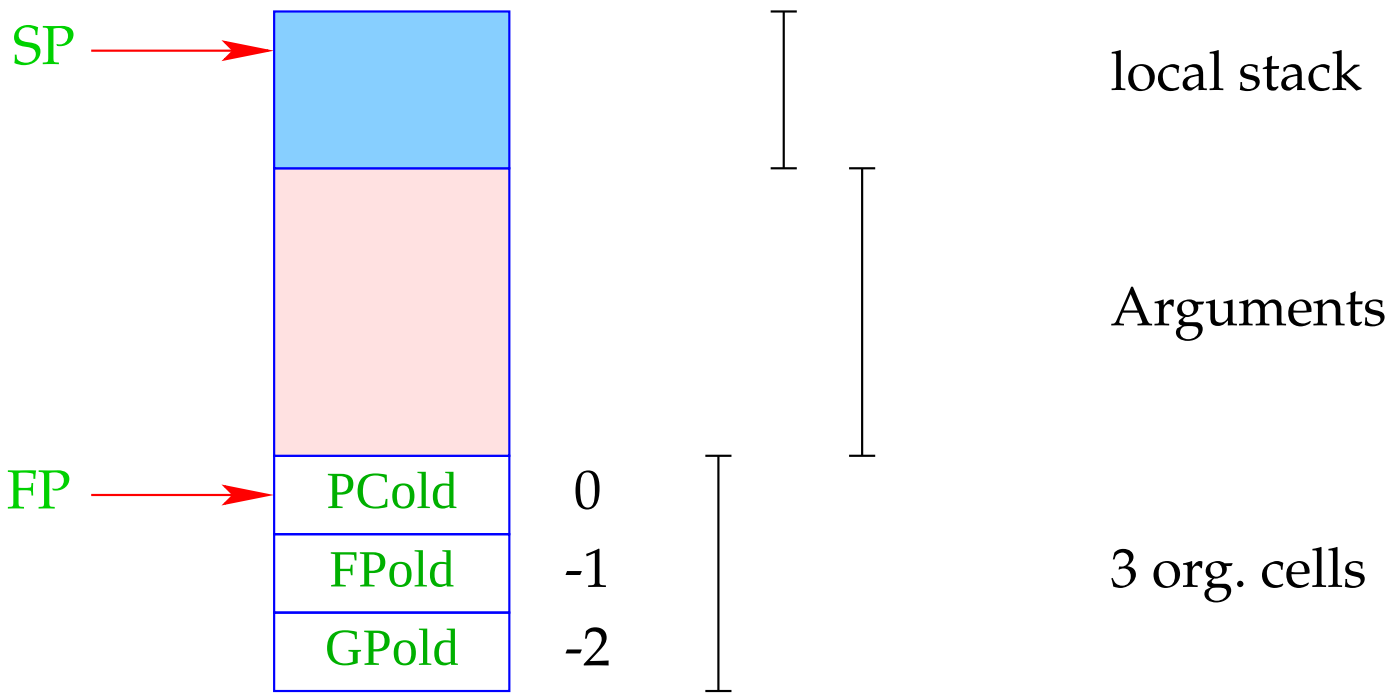
A Slightly Larger Example:

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

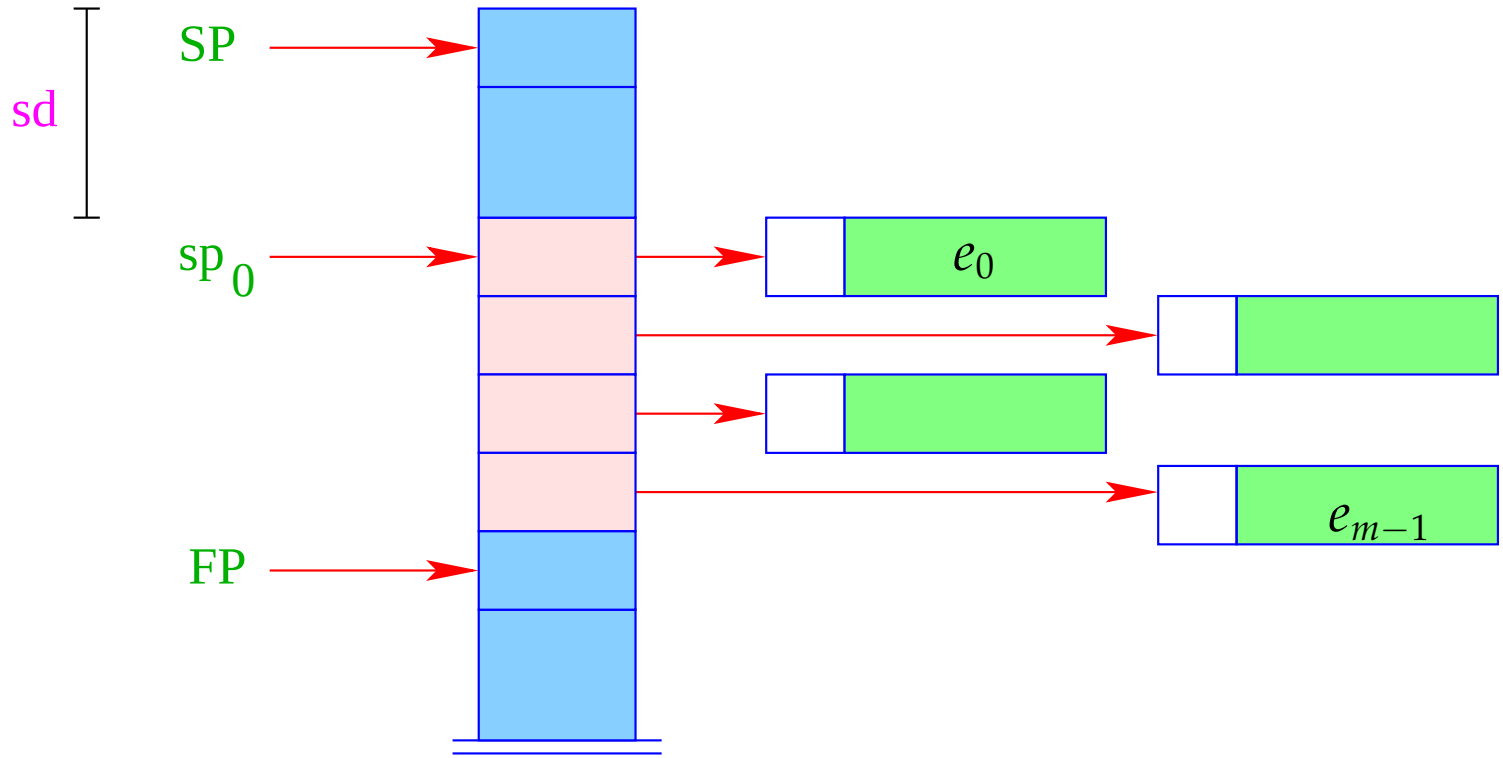
For CBV and $sd = 0$ we obtain:

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

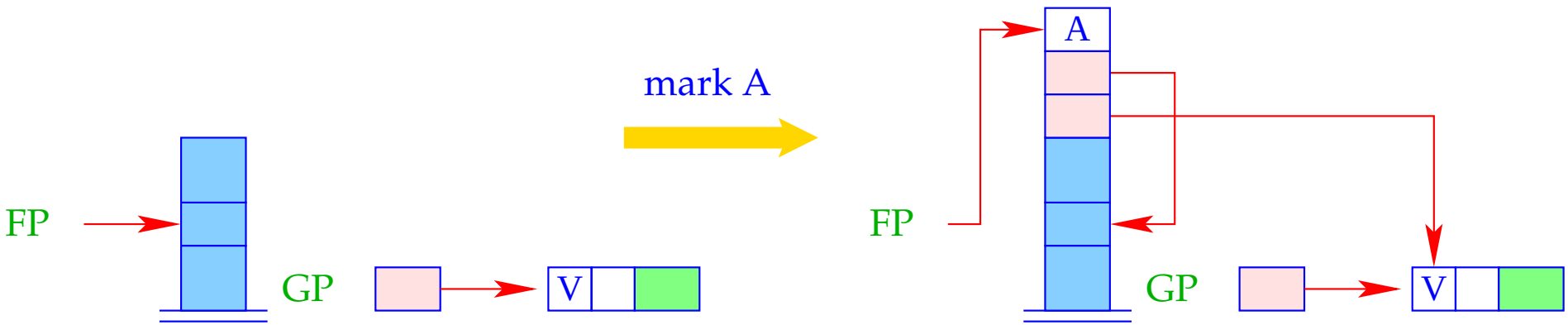
For the implementation of the new instruction, we must fix the organization of a stack frame:



Remember: Addressing of arguments and local variables

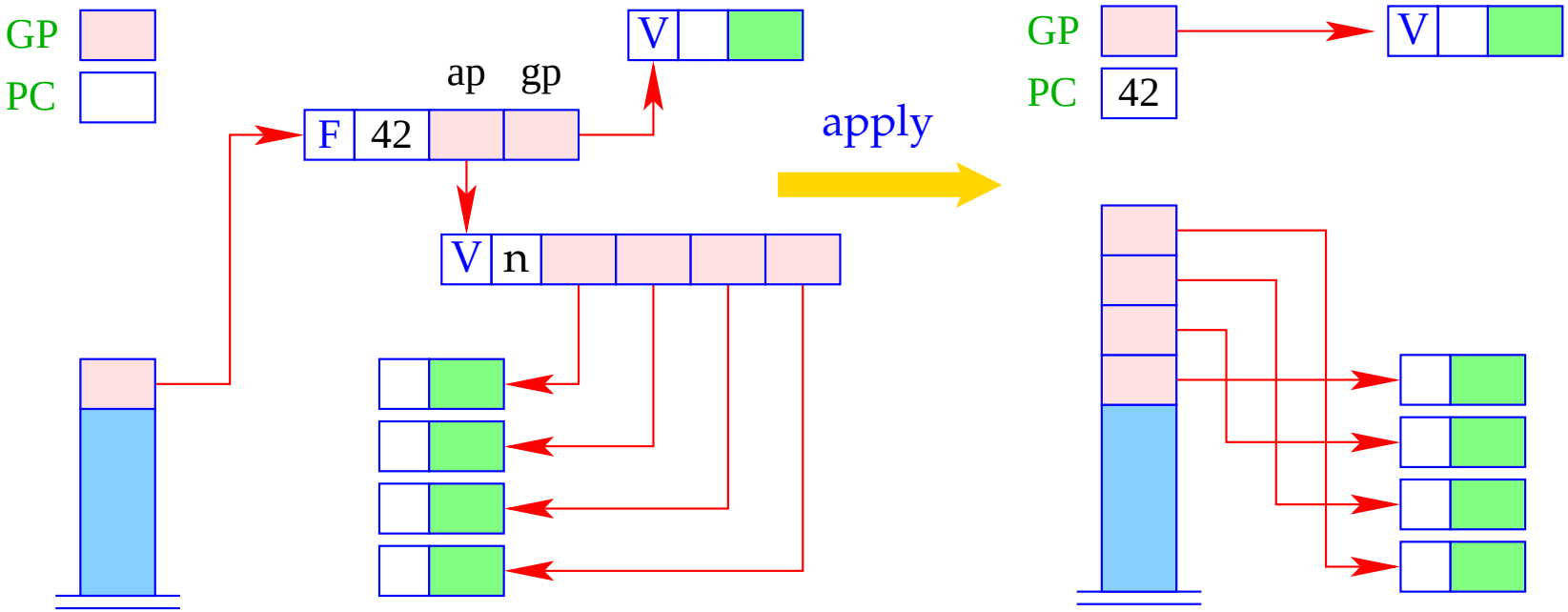


Different from the CMa, the instruction mark A already saves the return address:



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;
}
```

Warning:

- The last element of the argument vector is the last to be put onto the stack. This must be the first argument reference.
- This should be kept in mind, when we treat the packing of arguments of an under-supplied function application into an F-object !!!

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

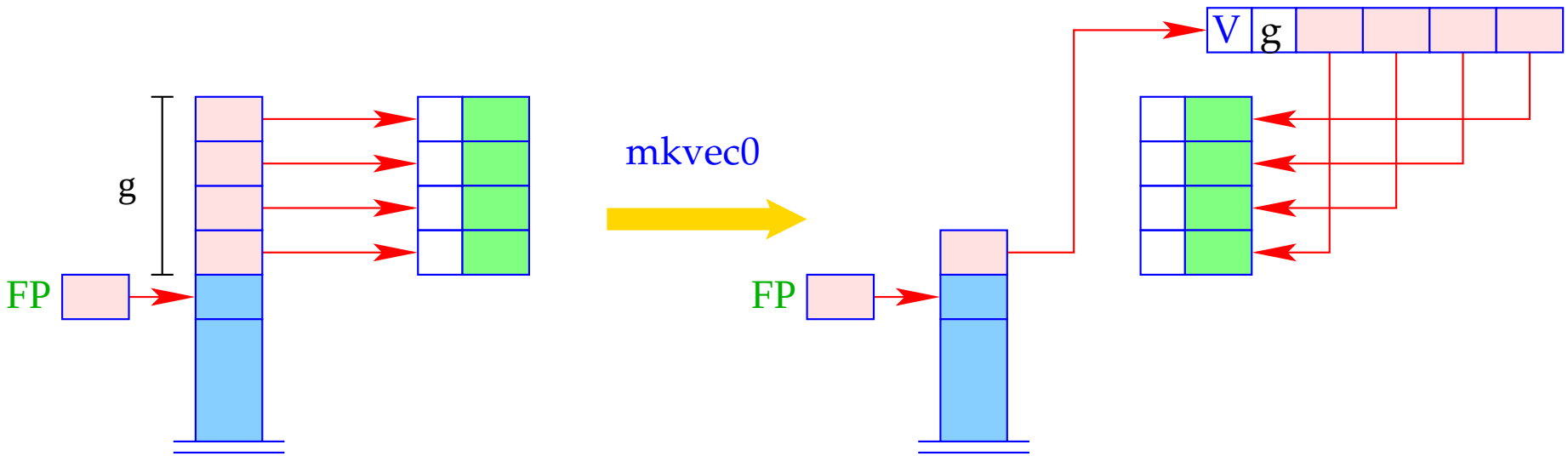
targ k is a complex instruction.

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
           }
```

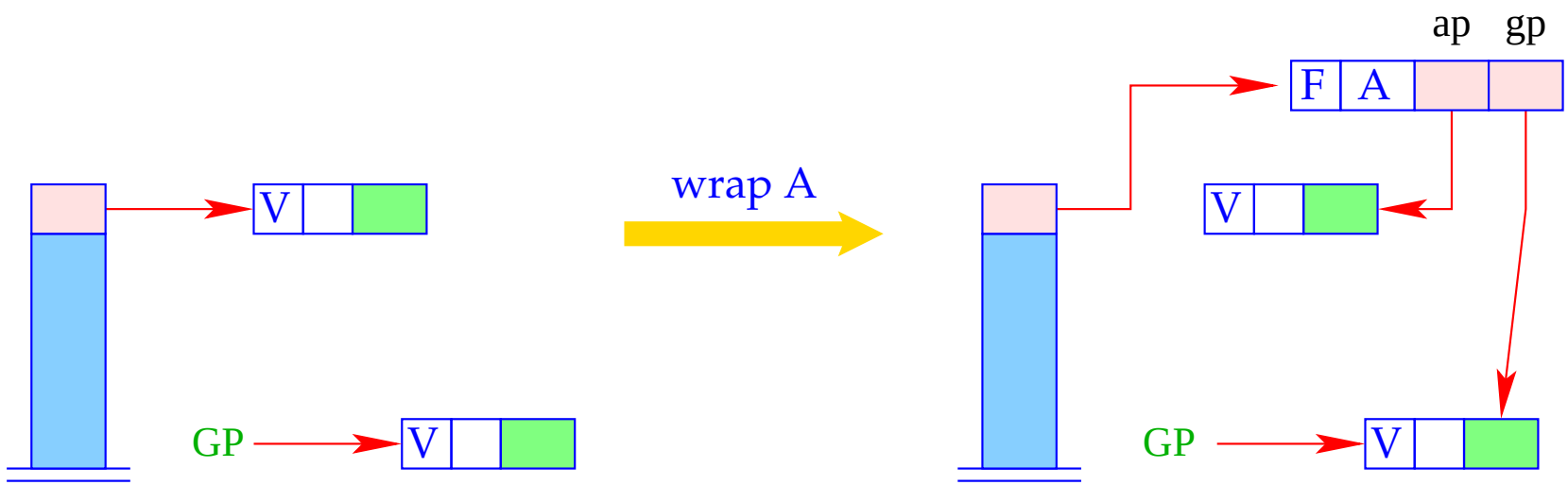
The combination of these steps into one instruction is a kind of optimization :-)

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



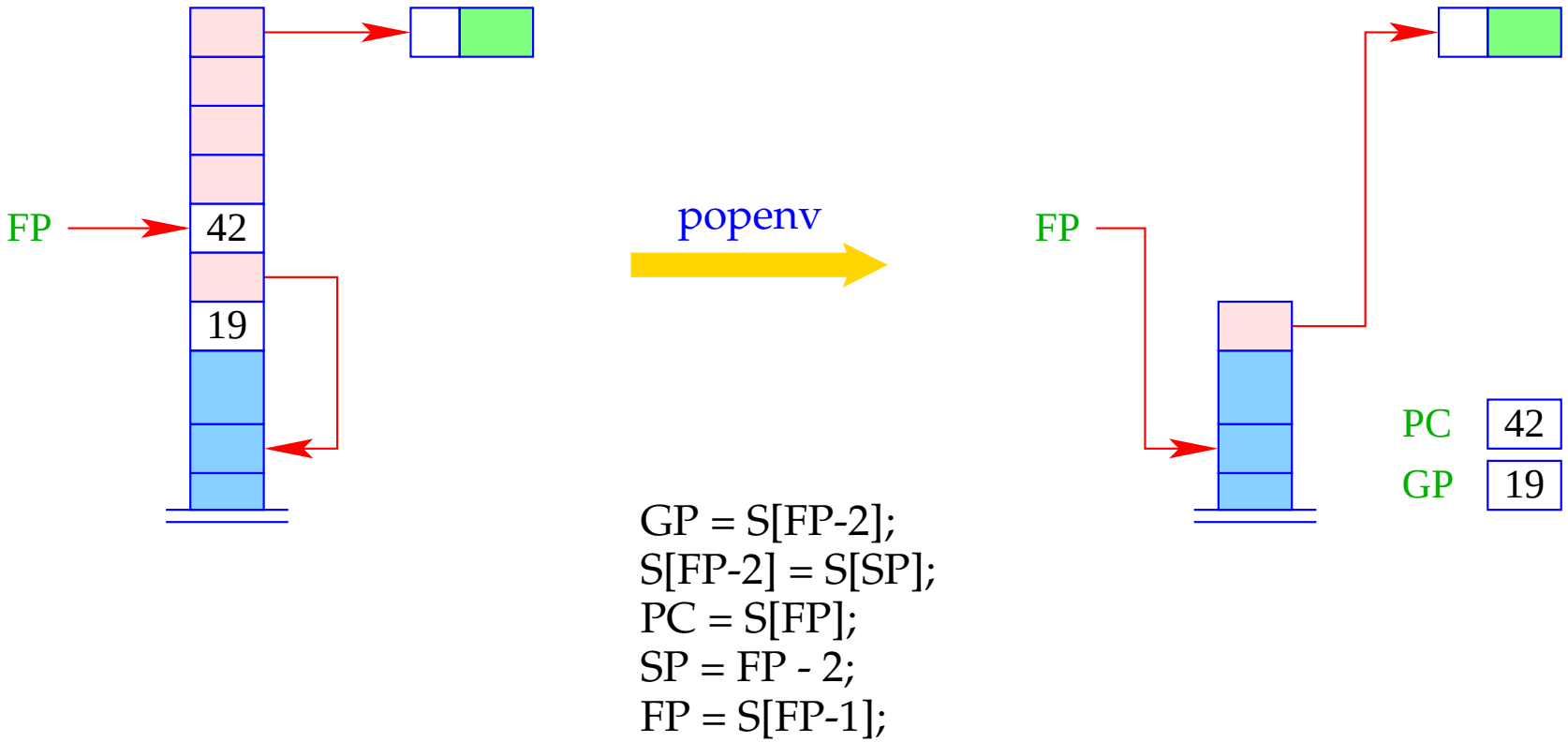

```
g = SP–FP; h = new (V, g);
SP = FP+1;
for (i=0; i<g; i++)
    h→v[i] = S[SP + i];
S[SP] = h;
```

The instruction wrap A wraps the argument vector together with the global vector into an F-object:



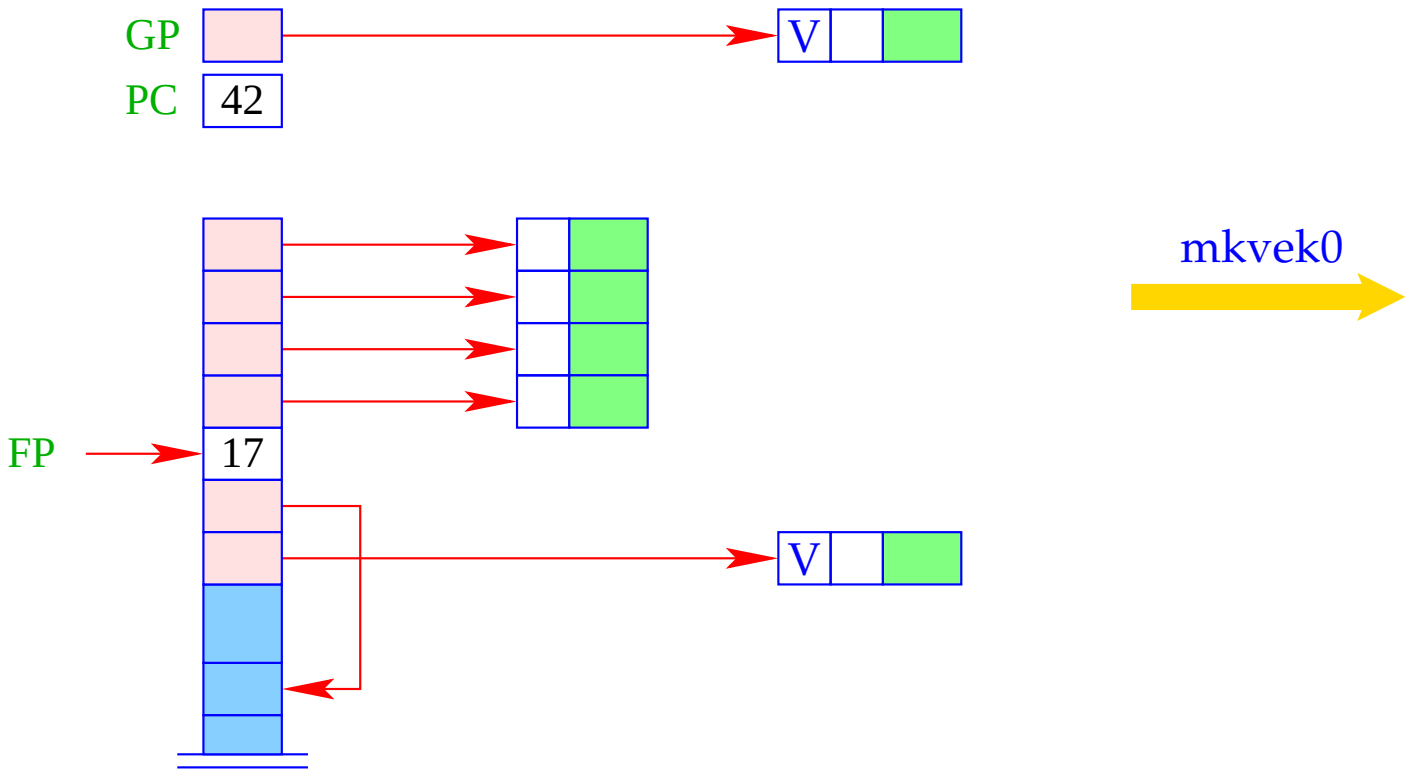
S[SP] = new (F, A, S[SP], GP);

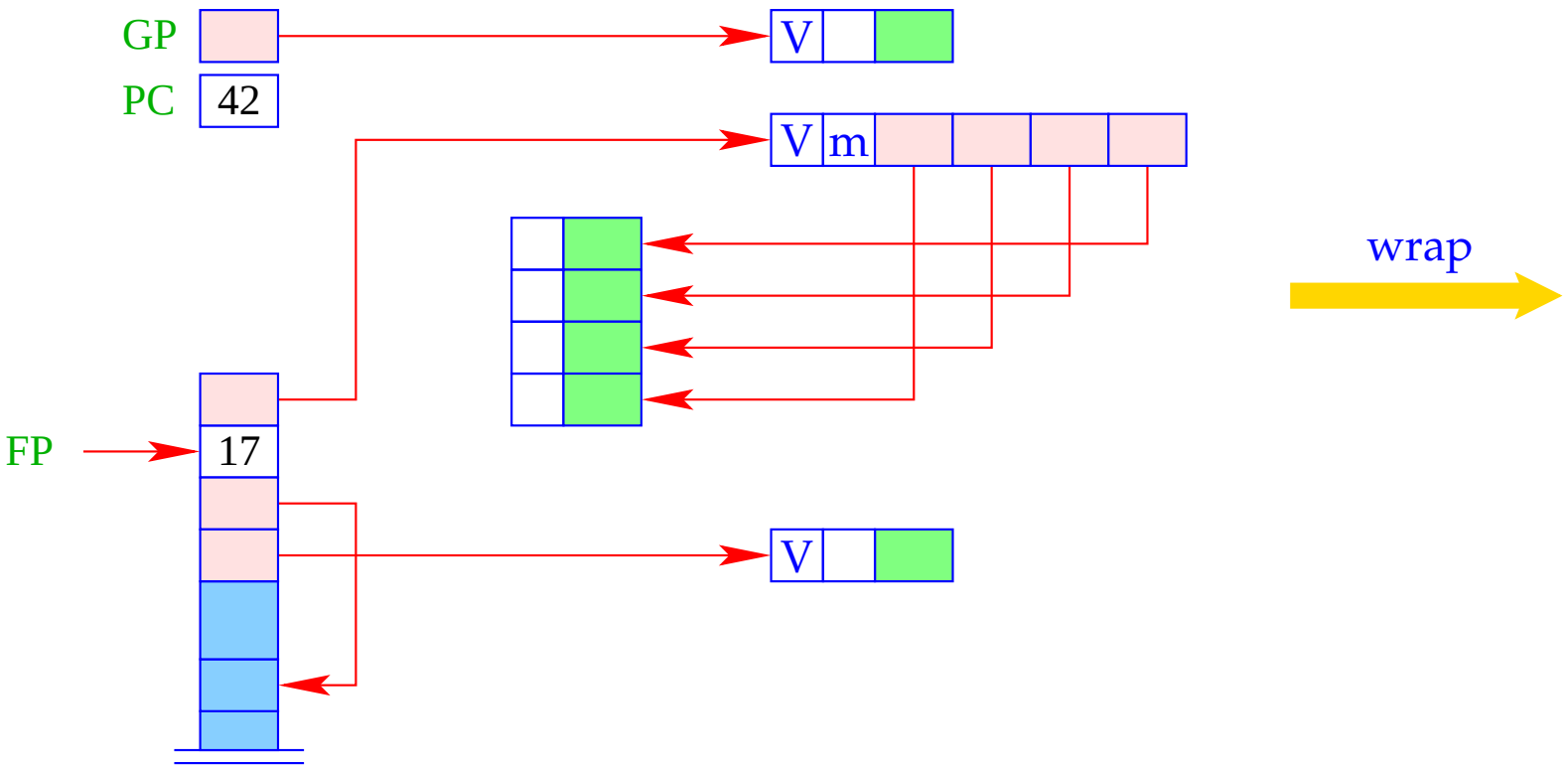
The instruction popenv finally releases the stack frame:

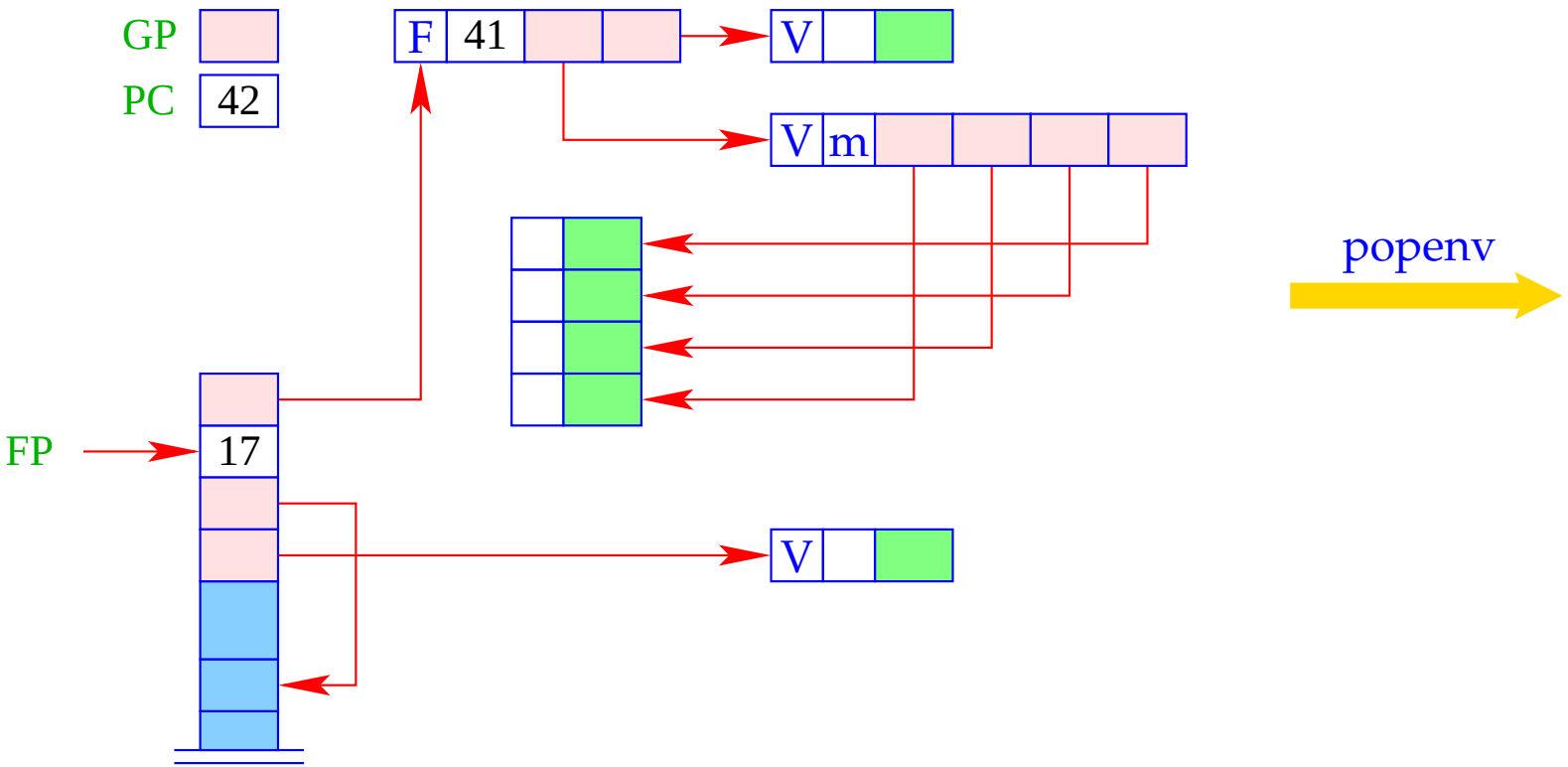


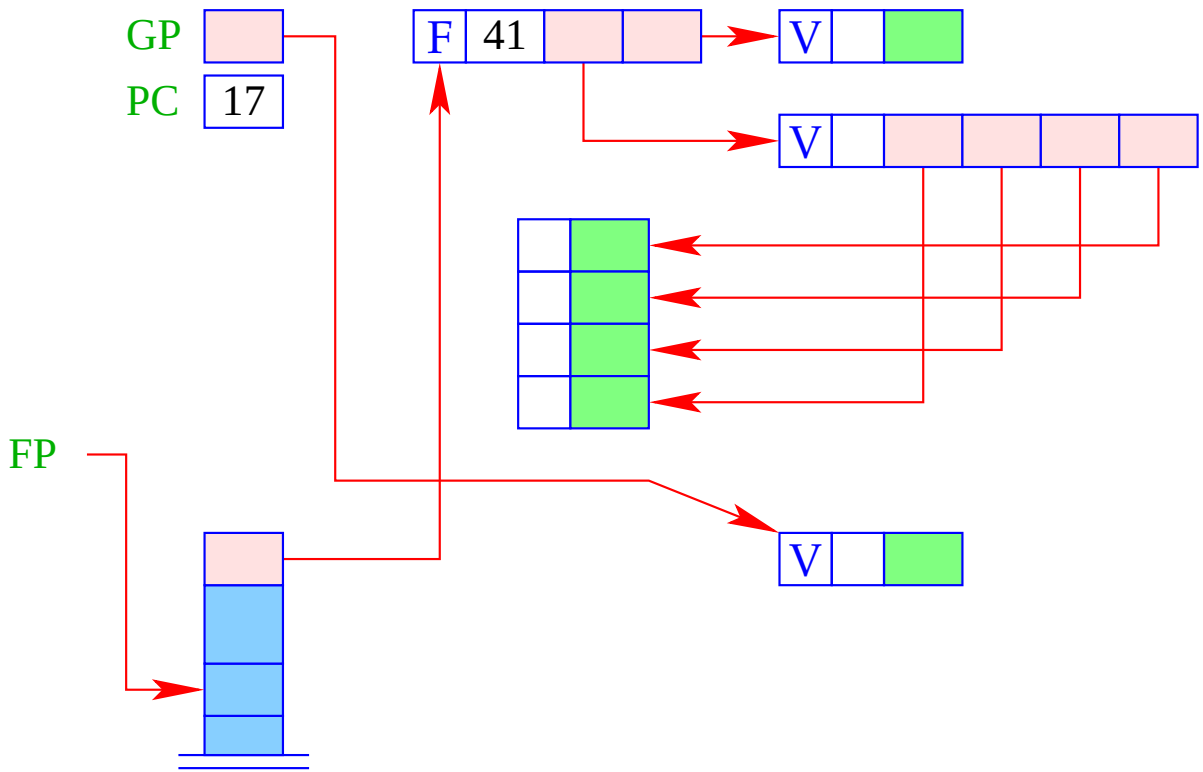

```
GP = S[FP-2];
S[FP-2] = S[SP];
PC = S[FP];
SP = FP - 2;
FP = S[FP-1];
```

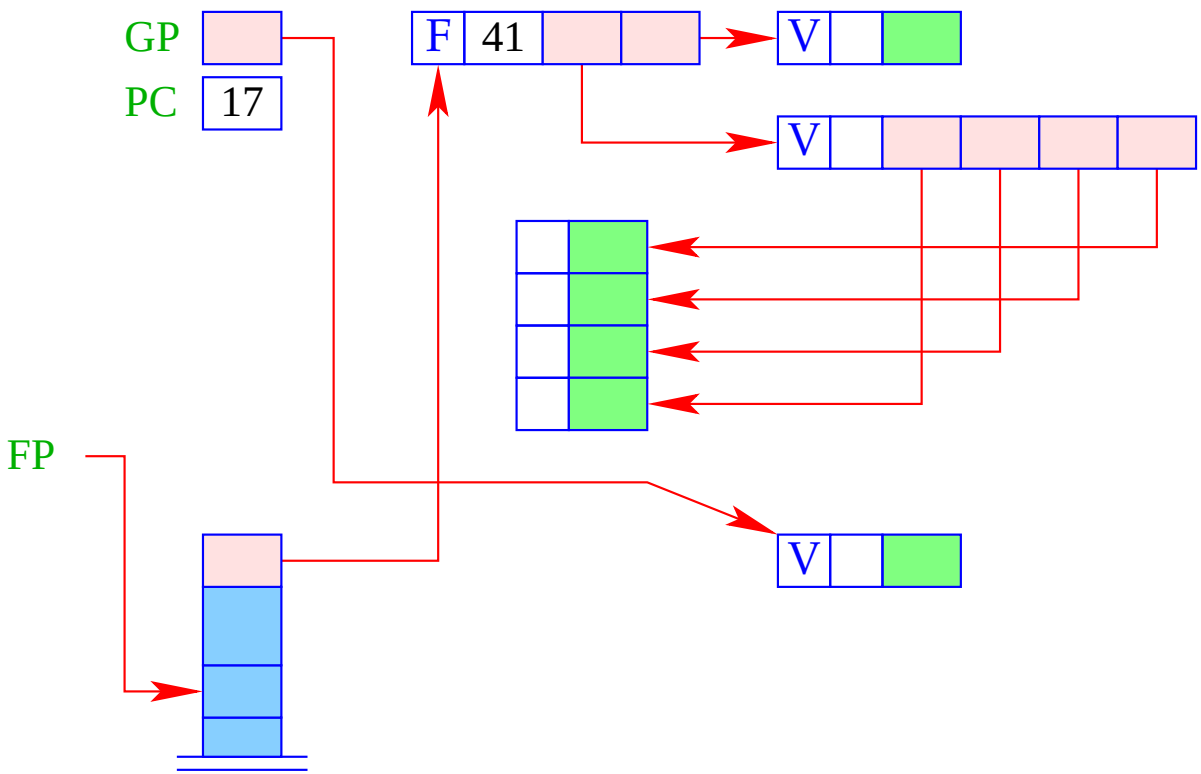
Thus, we obtain for targ k in the case of under supply:




GP
PC
42
V
V
m
wrap
FP
17
V


GP
PC
42
F
41
V
V
m
popenv
FP
17
V


GP
PC
17
F
41
V
V
V
FP

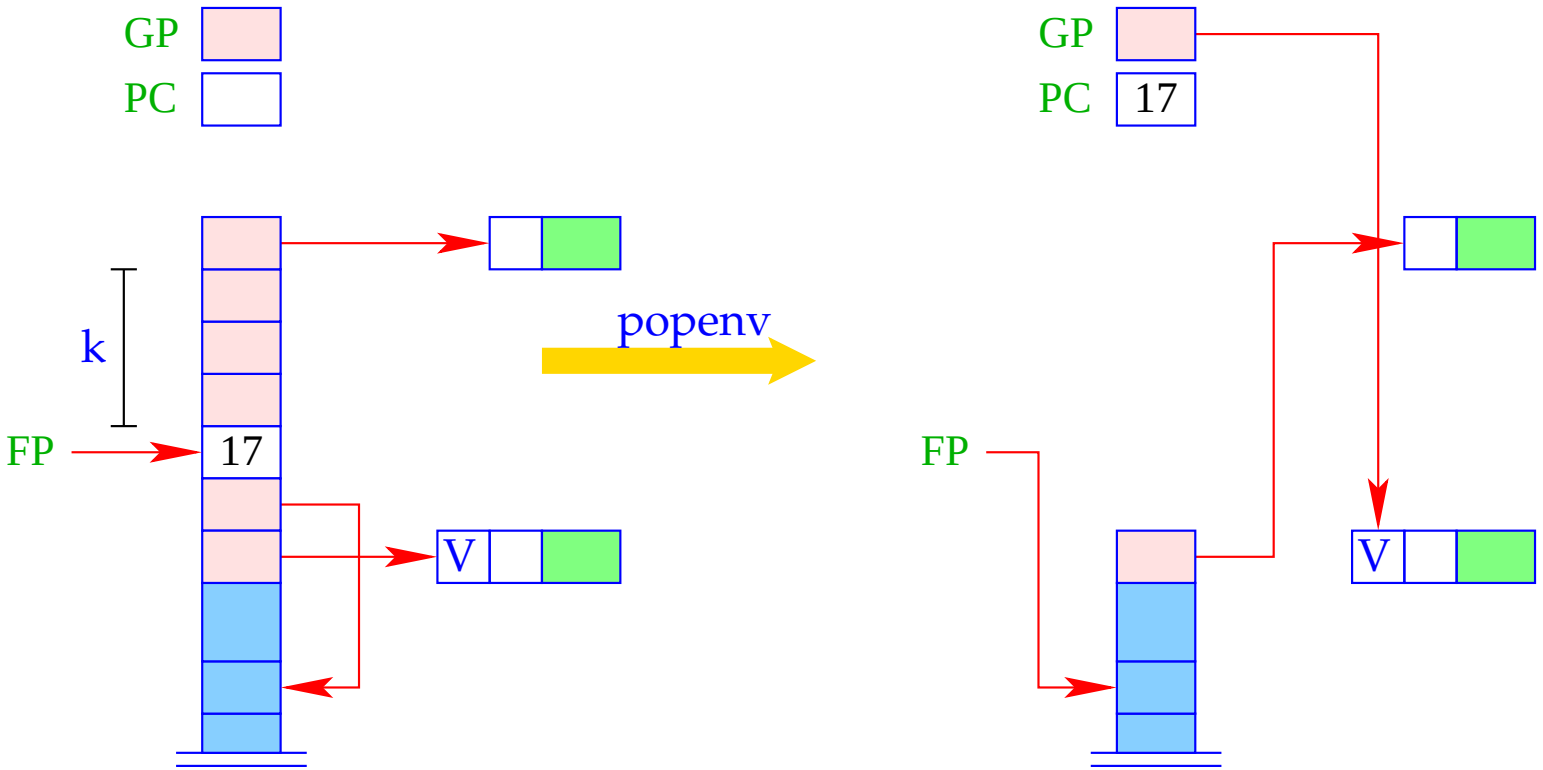

GP
PC
17
F
41
V
V
V
FP

- 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:

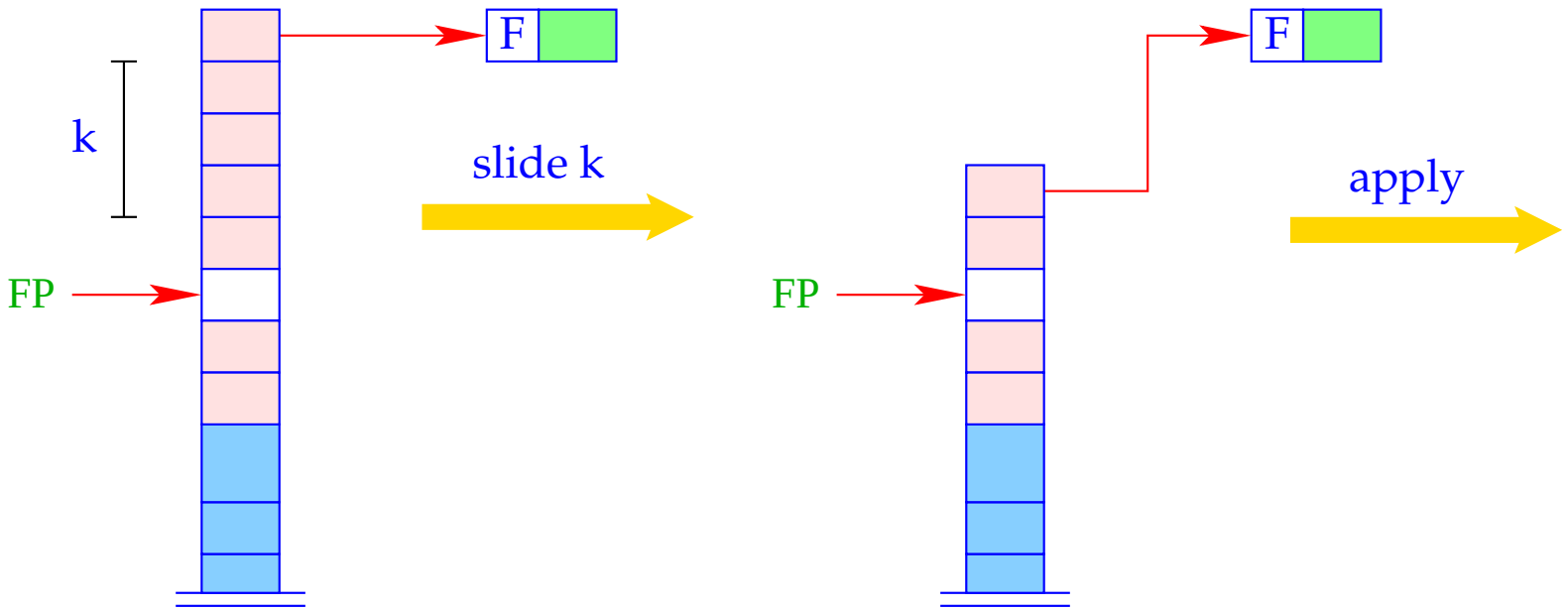
```
return k  =  if (SP − FP = k + 1)
               popenv;              // Done
             else {                 // There are more arguments
               slide k;
               apply;               // another application
             }
```

The execution of return k results in:

Case: Done



Case: Over-supply



19 letrec-Expressions

Consider the expression $e \equiv \textbf{letrec}\ y_1 = e_1; \ldots; y_n = e_n\ \textbf{in}\ e_0$.

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:

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

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!!!

Example:

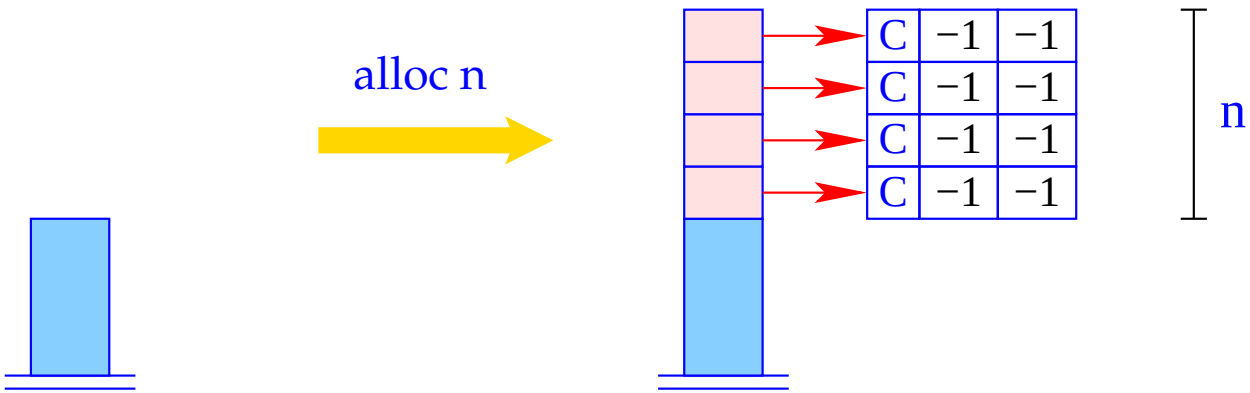
Consider the expression

$$e \equiv \textbf{letrec } f = \textbf{fn} x, y \Rightarrow \textbf{if} y \leq 1 \textbf{ then } x \textbf{ else } f(x * y)(y - 1) \textbf{ in } f1$$

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

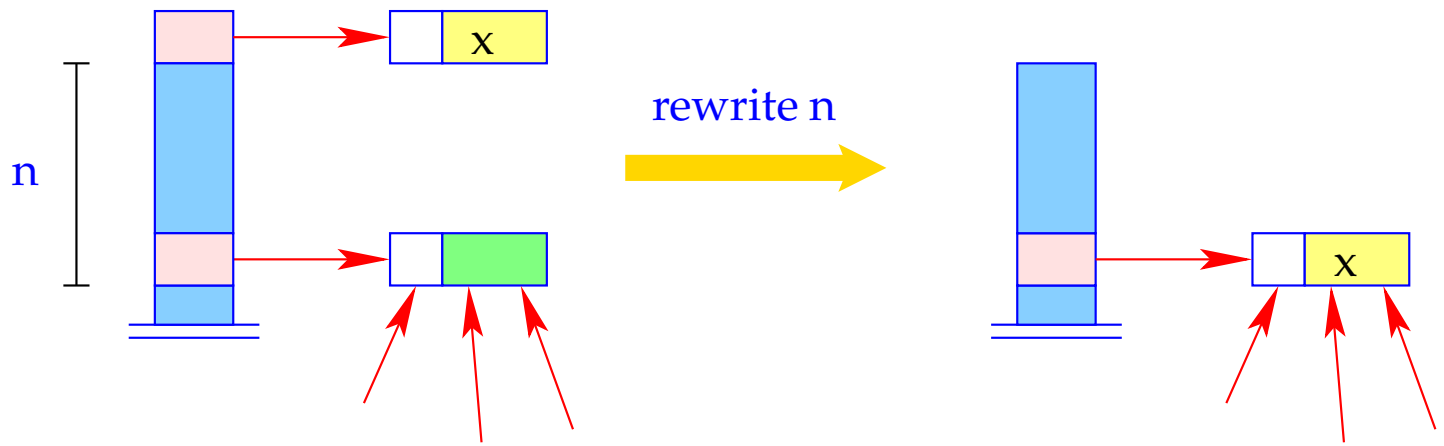
```
0   alloc 1          0   A:  targ 2          4       loadc 1
1   pushloc 0        0       ...             5       mkbasic
2   mkvec 1          1       return 2        5       pushloc 4
2   mkfunval A       2   B:  rewrite 1       6       apply
2   jump B           1       mark C          2   C: 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

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;
```

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!

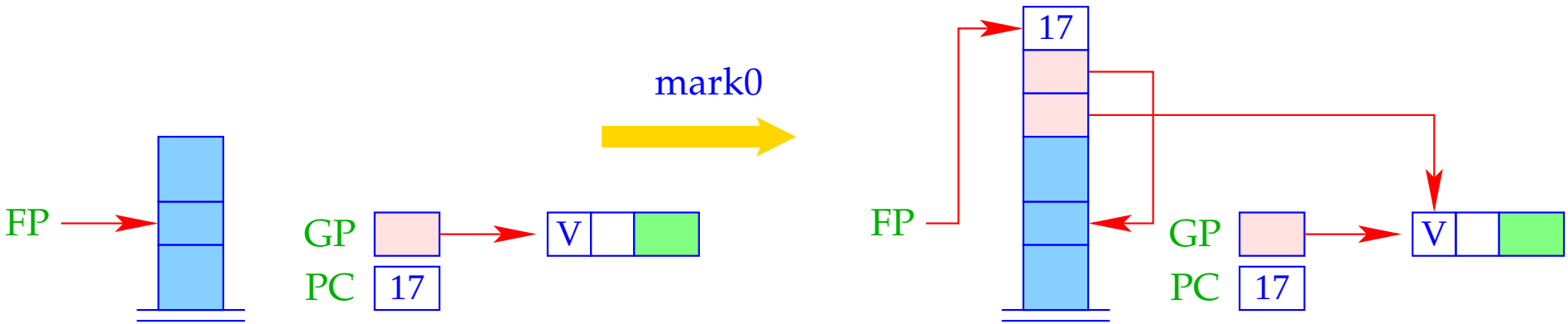
20 Closures and their Evaluation

- Closures are needed only for the implementation of CBN.
- 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.

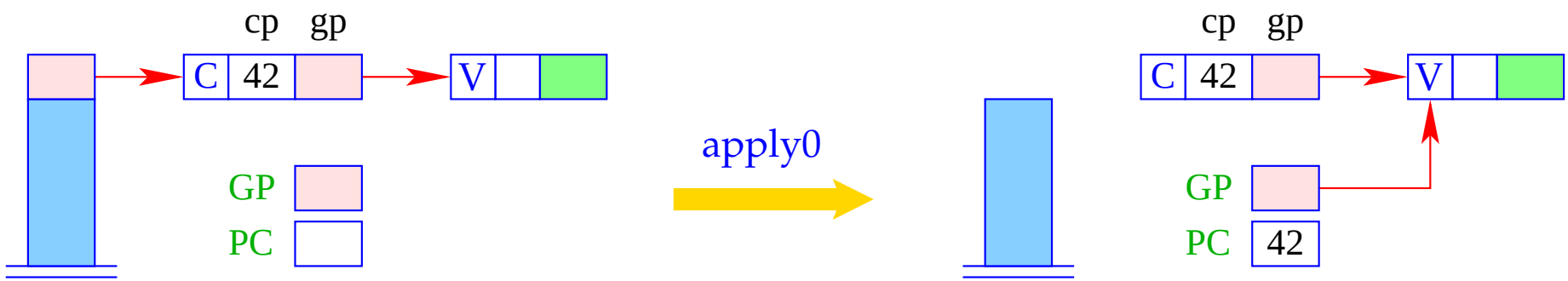
eval can be decomposed into small actions:

```
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.

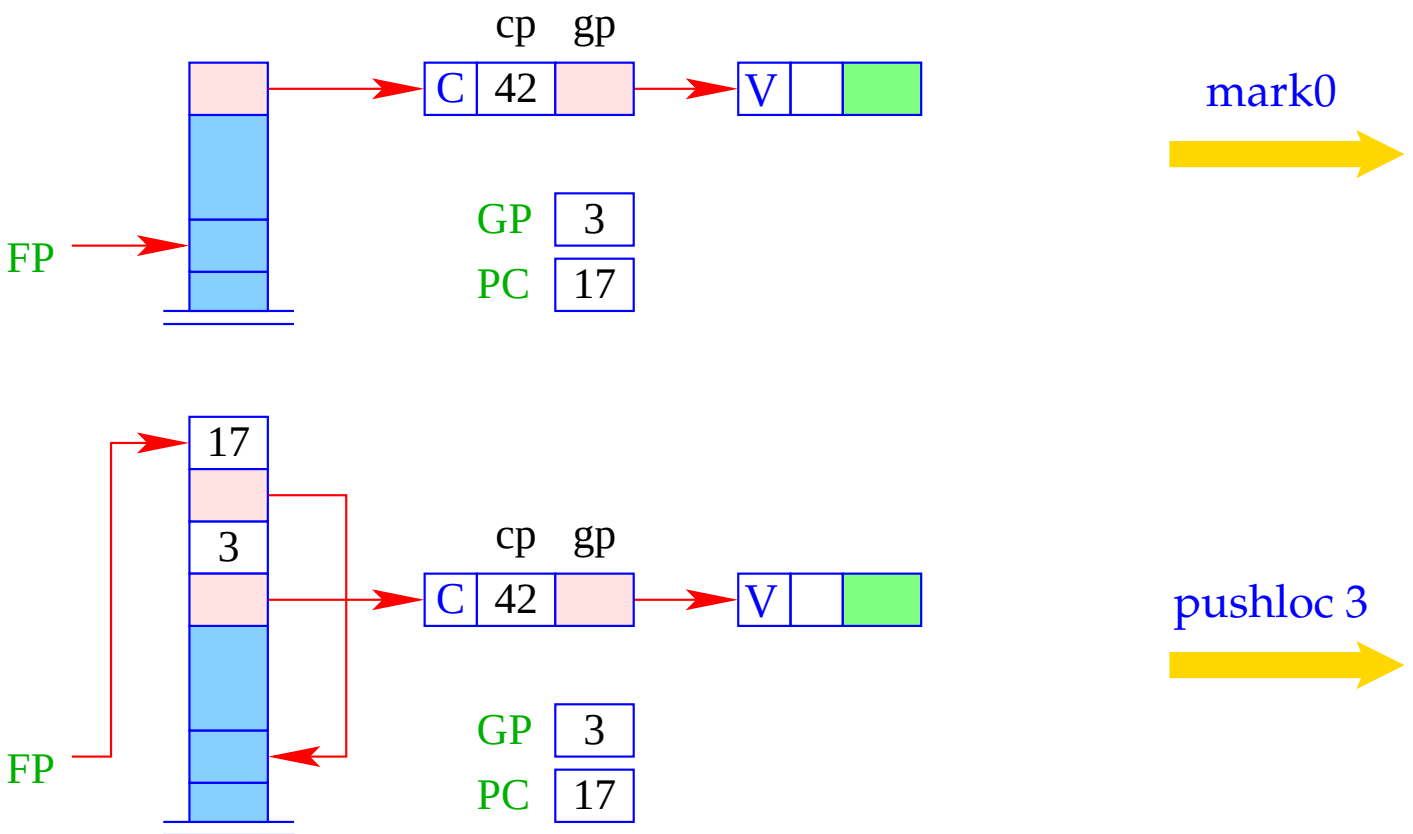


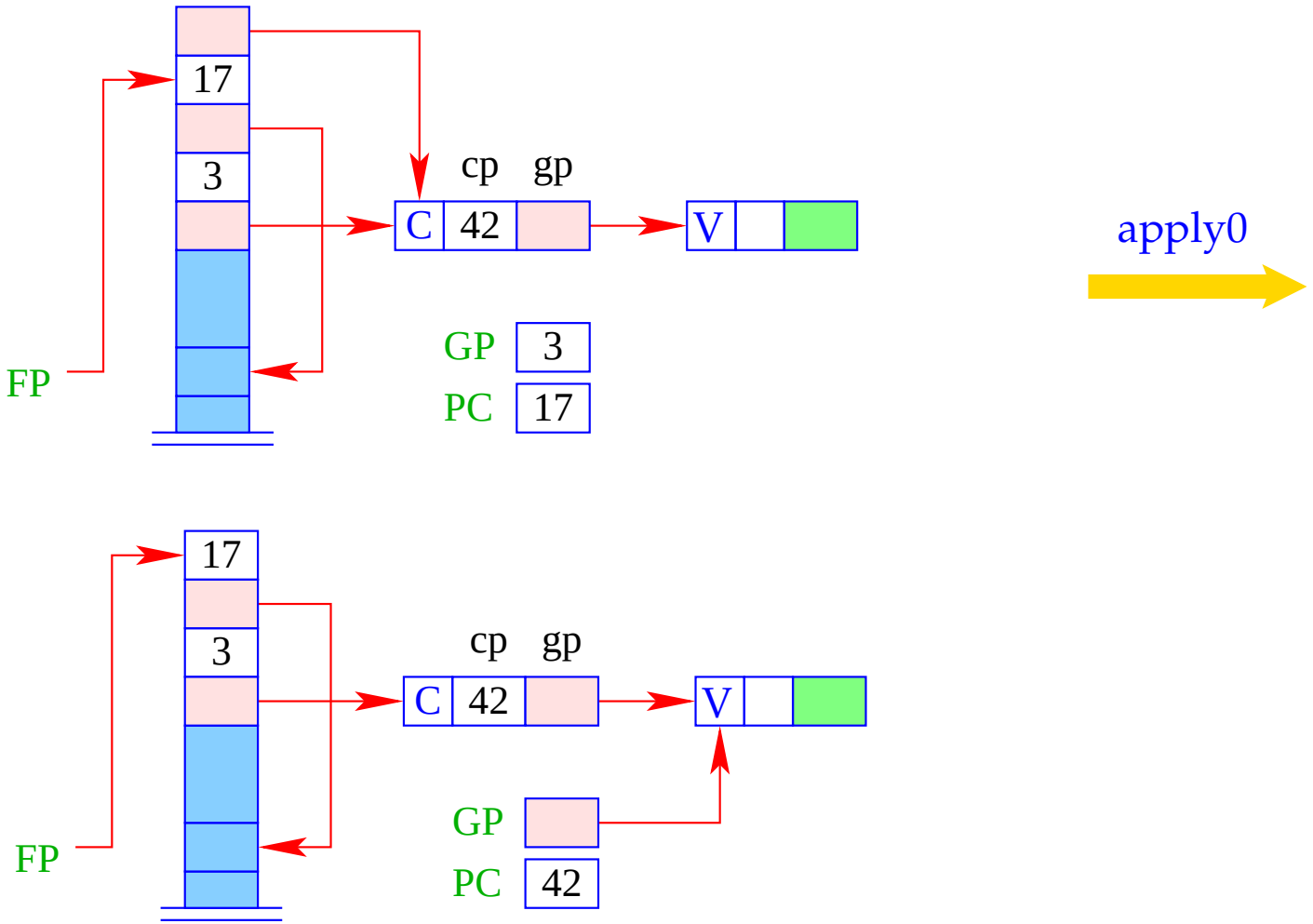
S[SP+1] = GP;
S[SP+2] = FP;
S[SP+3] = PC;
FP = SP = SP + 3;



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

We thus obtain for the instruction eval:


cp
gp
C
42
V
mark0
GP
3
PC
17
FP
17
3
cp
gp
C
42
V
pushloc 3
GP
3
PC
17
FP


17
3
cp
gp
C
42
V
FP
GP
3
PC
17
apply0
17
3
cp
gp
C
42
V
FP
GP
PC
42

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:

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

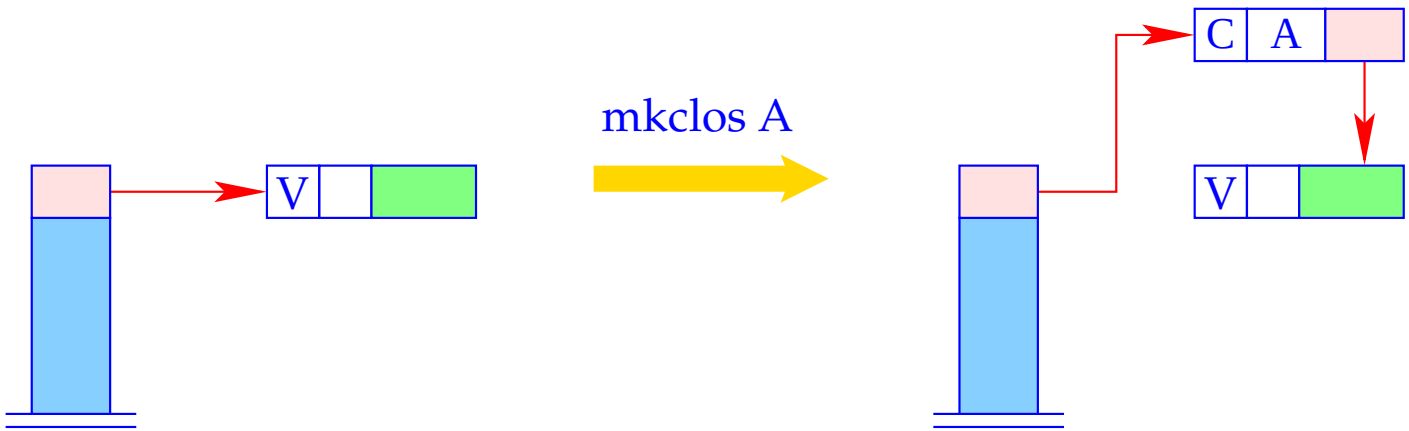
where $\{z_0, \ldots, z_{g-1}\} = \mathit{free}(e)$ and $\rho' = \{z_i \mapsto (G, i) \mid i = 0, \ldots, g-1\}$.

Example:

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

- The instruction mkclos A is analogous to the instruction mkfunval A.
- It generates a C-object, where the included code pointer is A.



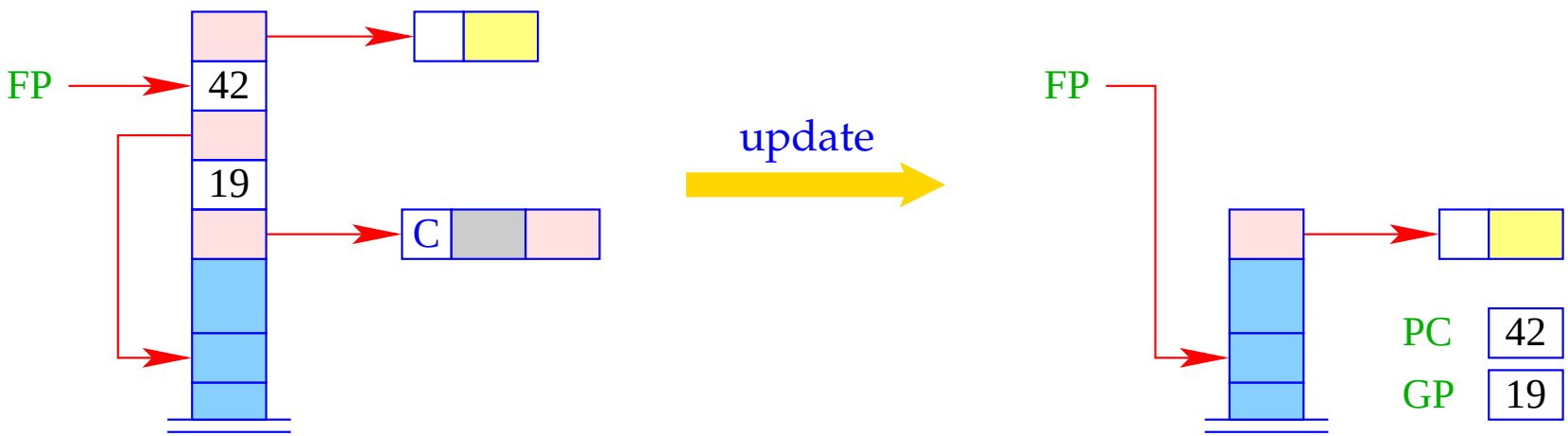
S[SP] = new (C, A, S[SP]);

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:

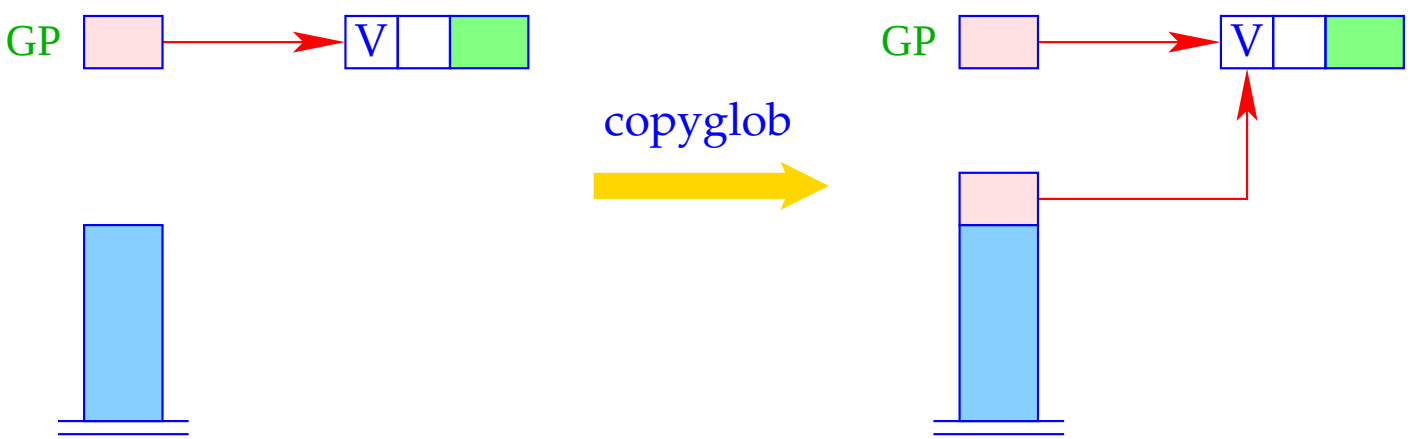
- Functional programs construct many F- and C-objects.
- This requires the inclusion of (the bindings of) all global variables.
  Recall, e.g., the construction of a closure for an expression e ...

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

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

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 :



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:

$e \equiv$ **letrec** $a = b; b = 7$ **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

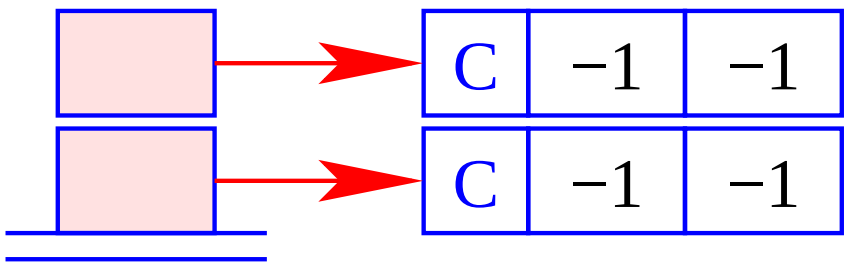
The execution of this instruction sequence should deliver the basic value 7 ...

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

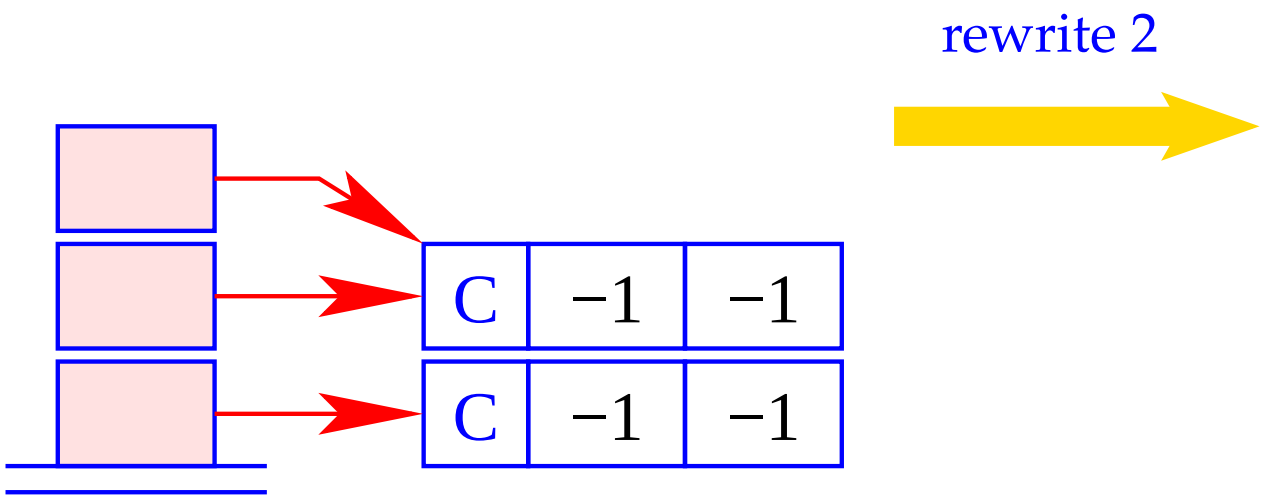
alloc 2

```
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
```

pushloc 0

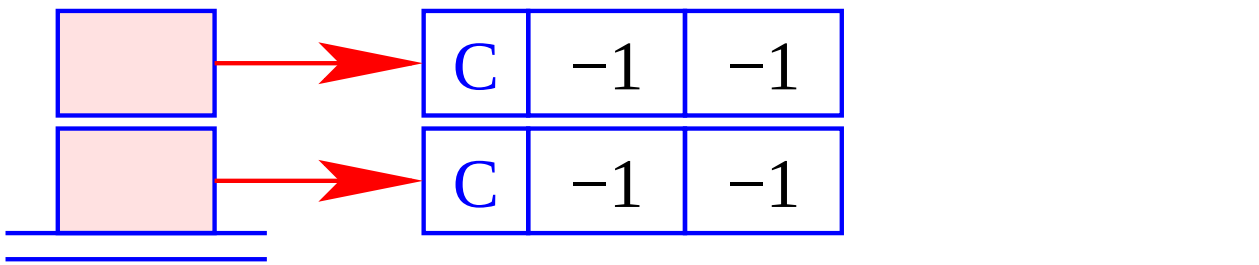


```
0  alloc 2        3  rewrite 2      3  mkbasic        2  pushloc 1
2  pushloc 0      2  loadc 7        3  rewrite 1      3  eval
                                                     3  slide 2
```

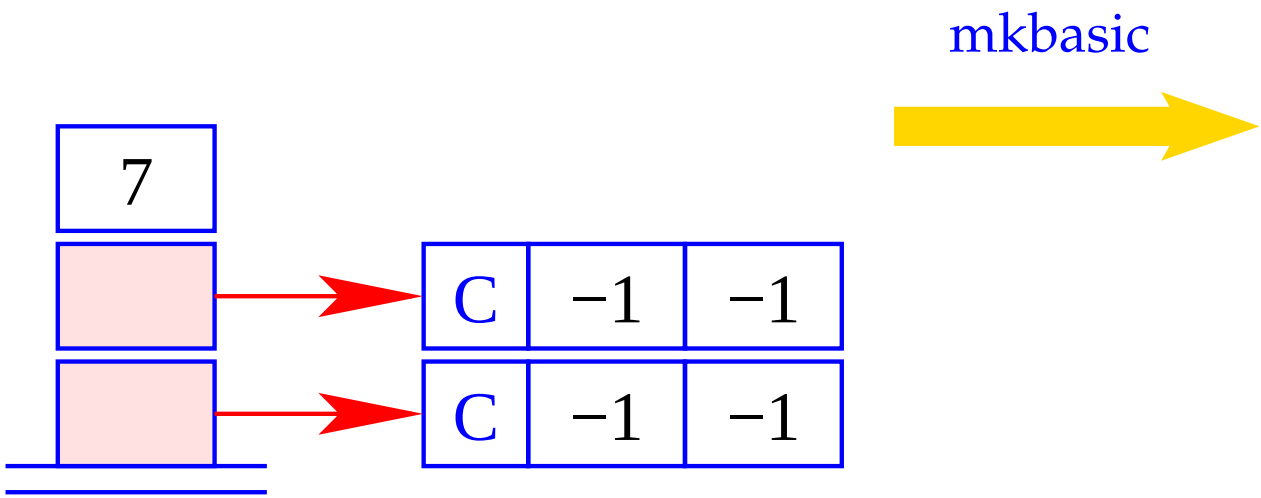


0	alloc 2	3	rewrite 2	3	mkbasic	2	pushloc 1
2	pushloc 0	2	loadc 7	3	rewrite 1	3	eval
						3	slide 2

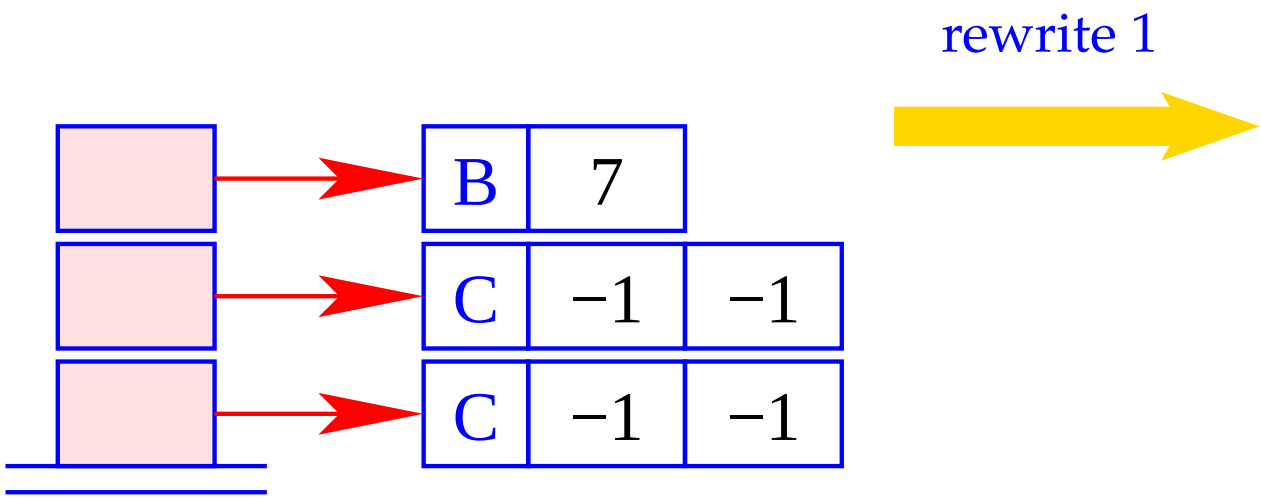
loadc 7



```
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
```

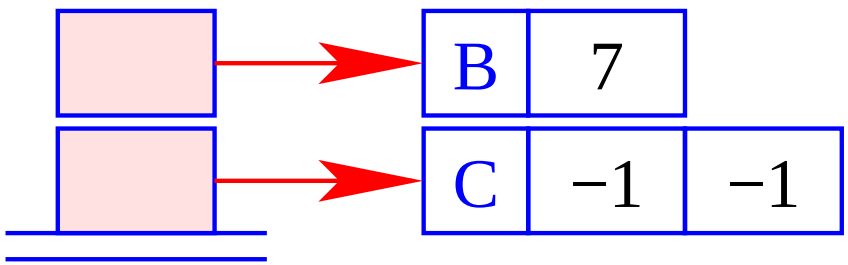


```
0  alloc 2        3  rewrite 2      3  mkbasic        2  pushloc 1
2  pushloc 0      2  loadc 7        3  rewrite 1      3  eval
                                                     3  slide 2
```

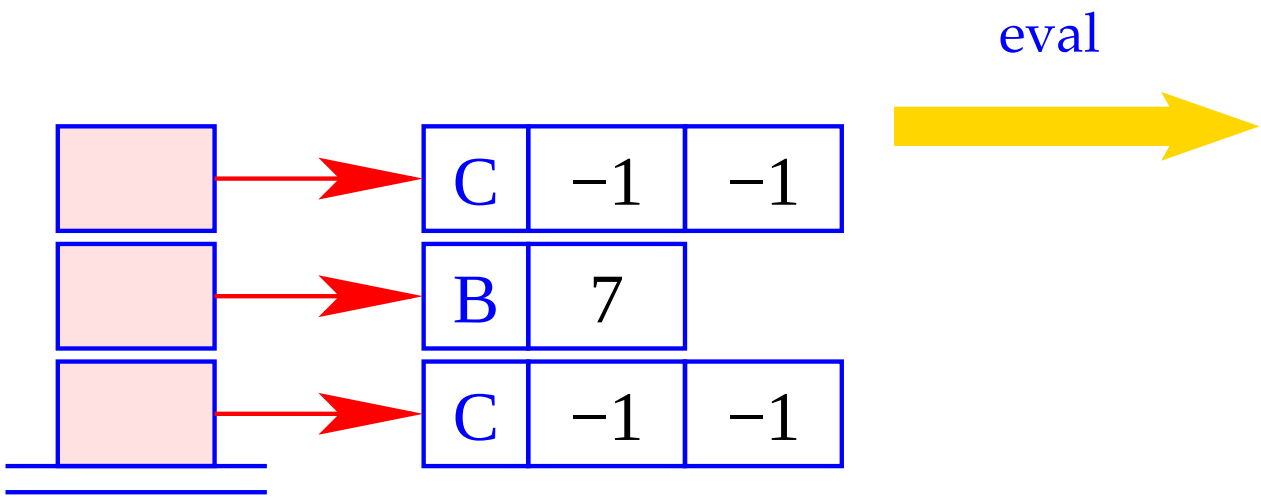


```
0  alloc 2        3  rewrite 2      3  mkbasic        2  pushloc 1
2  pushloc 0      2  loadc 7        3  rewrite 1      3  eval
                                                     3  slide 2
```

pushloc 1



```
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
```

Segmentation Fault !!

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** ....

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

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:

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

23 The Translation of a Program Expression

Execution of a program e starts with

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

The expression e must not contain free variables.

The value of e should be determined and then a halt instruction should be executed.

$$\begin{array}{lcl} \mathrm{code}\ e & = & \mathrm{code}_V\ e\ \emptyset\ 0 \\ & & \mathrm{halt} \end{array}$$

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

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
```

24 Structured Data

In the following, we extend our functional programming language by some datatypes.

24.1 Tuples

Constructors: $(.,\ldots,.)$, k-ary with $k \geq 0$;

Destructors: $\#j$ for $j \in \mathbb{N}_0$ (Projections)

We extend the syntax of expressions correspondingly:

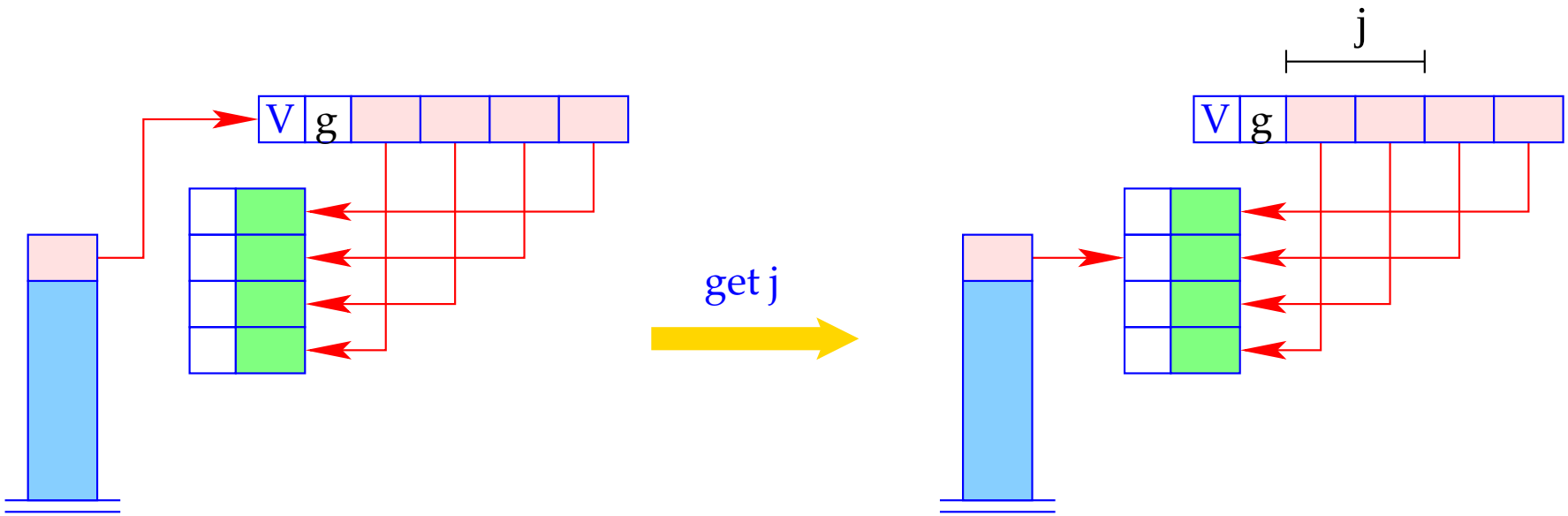
$$\begin{aligned} e \quad ::= \quad & \ldots \mid (e_0,\ldots,e_{k-1}) \mid \#j\ e \\ & \mid \textbf{let}\ (x_0,\ldots,x_{k-1}) = e_1\ \textbf{in}\ e_0 \end{aligned}$$

- In order to construct a tuple, we collect sequence of references on the stack. Then we construct a vector of these references in the heap using mkvec
- For returning components we use an indexed access into the tuple.

$$
\begin{array}{lcl}
\text{code}_V\ (e_0, \ldots, e_{k-1})\ \rho\ \text{sd} & = & \text{code}_C\ e_0\ \rho\ \text{sd} \\
 & & \text{code}_C\ e_1\ \rho\ (\text{sd}+1) \\
 & & \ldots \\
 & & \text{code}_C\ e_{k-1}\ \rho\ (\text{sd}+k-1) \\
 & & \text{mkvec k}
\end{array}
$$

$$
\begin{array}{lcl}
\text{code}_V\ (\#j\ e)\ \rho\ \text{sd} & = & \text{code}_V\ e\ \rho\ \text{sd} \\
 & & \text{get j}
\end{array}
$$

In the case of CBV, we directly compute the values of the e_i.




```
if (S[SP] == (V,g,v))
   S[SP] = v[j];
else Error “Vector expected!”;
```

Inversion: Accessing all components of a tuple simulataneously:

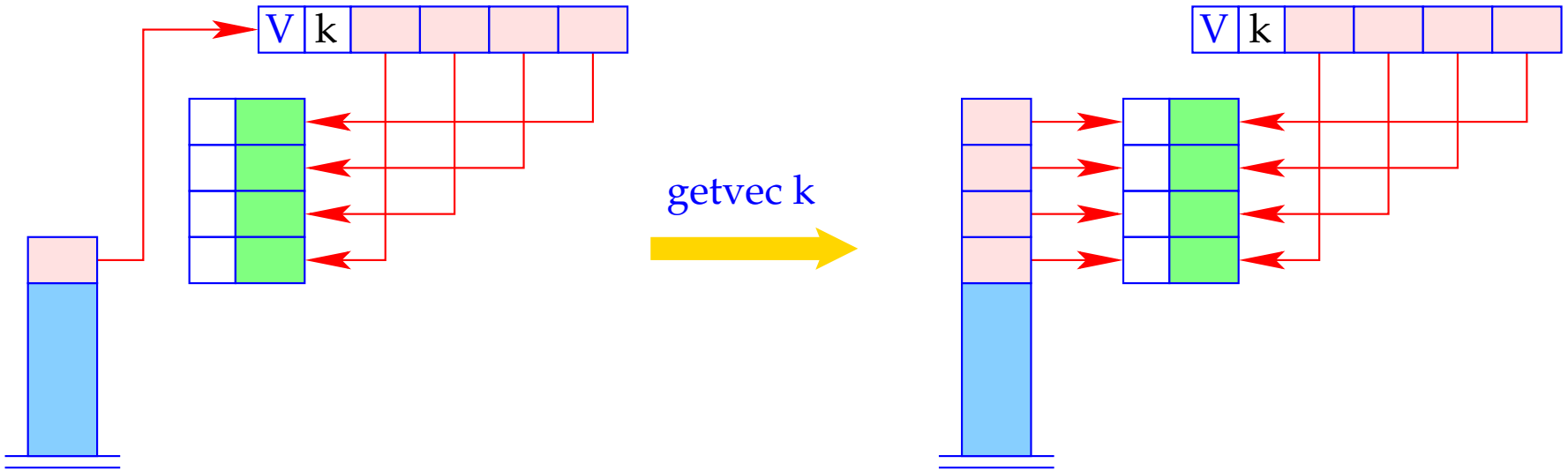
$$e \equiv \textbf{let } (y_0, \ldots, y_{k-1}) = e_1 \textbf{ in } e_0$$

This is translated as follows:

$$\begin{array}{lcl} \text{code}_V\ e\ \rho\ \text{sd} & = & \text{code}_V\ e_1\ \rho\ \text{sd} \\ & & \text{getvec k} \\ & & \text{code}_V\ e_0\ \rho'\ (\text{sd} + k) \\ & & \text{slide k} \end{array}$$

where $\rho' = \rho \oplus \{y_i \mapsto (L, sd + i) \mid i = 0, \ldots, k-1\}$.

The instruction getvec k pushes the components of a vector of length k onto the stack:




```
if (S[SP] == (V,k,v)) {
   SP--;
   for(i=0; i<k; i++) {
        SP++; S[SP] = v[i];
        }
} else Error “Vector expected!”;
```

24.2 Lists

Lists are constructed by the constructors:

[ ] "Nil", the empty list;

":" "Cons", right-associative, takes an element and a list.

Access to list components is possible by **case**-expressions ...

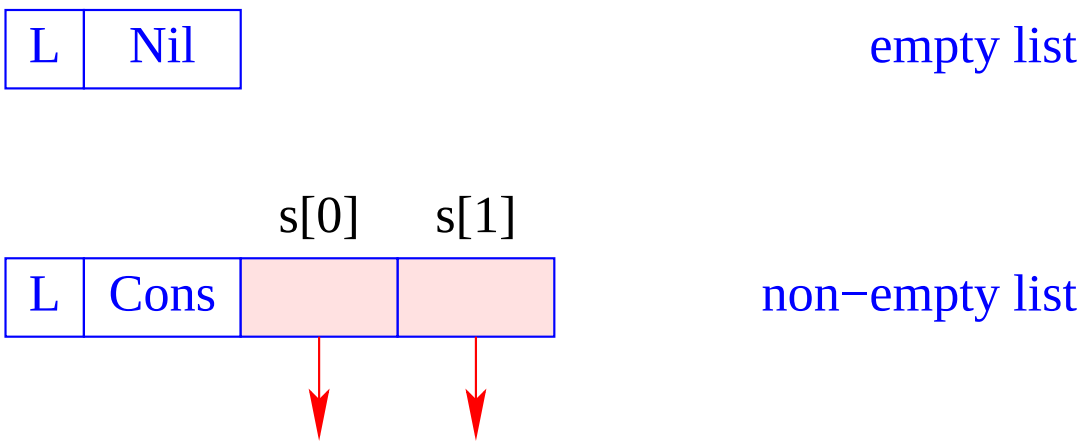
Example: The append function app:

$$
\begin{array}{lll}
\text{app} & = & \textbf{fn}\ l, y \Rightarrow \textbf{case}\ l\ \textbf{of} \\
 & & \quad [] \quad \rightarrow \quad y \\
 & & \quad h : t \quad \rightarrow \quad h : (\text{app}\ t\ y)
\end{array}
$$

accordingly, we extend the syntax of expressions:

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

Additionally, we need new heap objects:



24.3 Building Lists

The new instructions nil and cons are introduced for building list nodes.

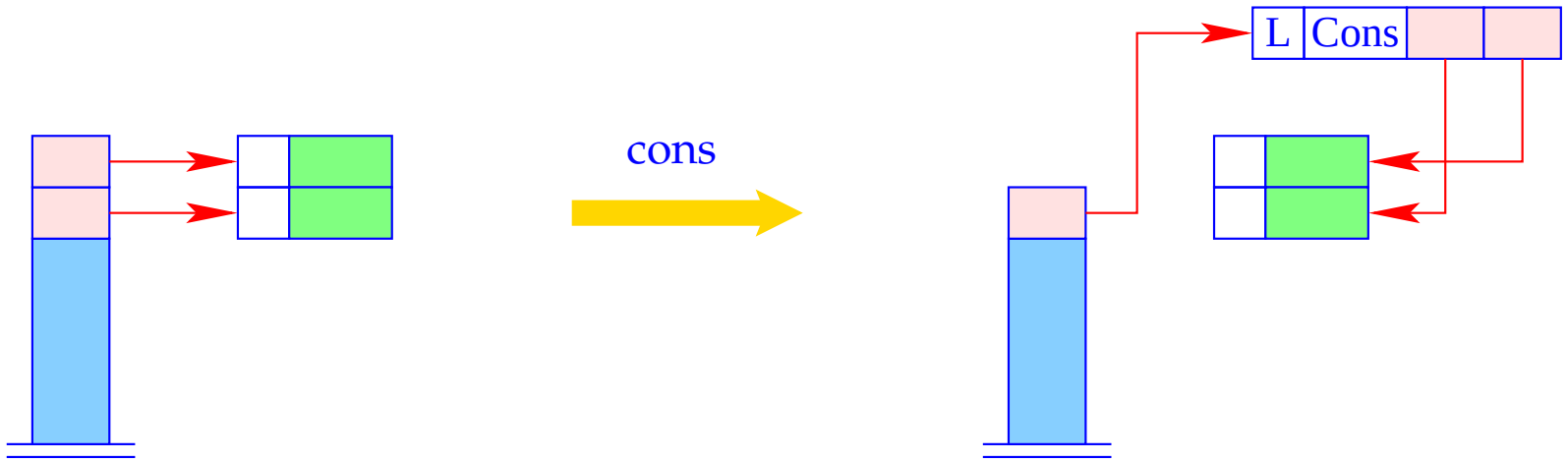
We translate for CBN:

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

Note:

- With CBN: Closures are constructed for the arguments of ":";
- With CBV: Arguments of ":" are evaluated :-)

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



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

24.4 Pattern Matching

Consider the expression $e \equiv \textbf{case}\ e_0\ \textbf{of}\ [] \rightarrow e_1;\ \ h : t \rightarrow e_2$.

Evaluation of e requires:

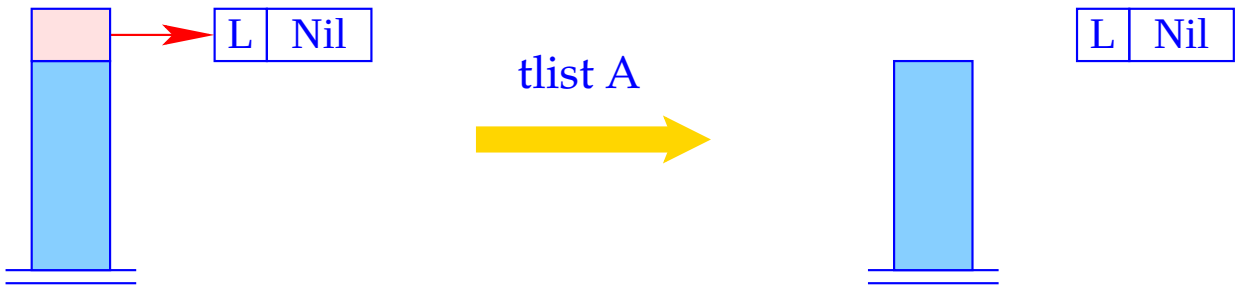
- evaluation of e_0;
- check, whether resulting value v is an L-object;
- if v is the empty list, evaluation of e_1 ...
- otherwise storing the two references of v on the stack and evaluation of e_2. This corresponds to binding h and t to the two components of v.

In consequence, we obtain (for CBN as for CBV):

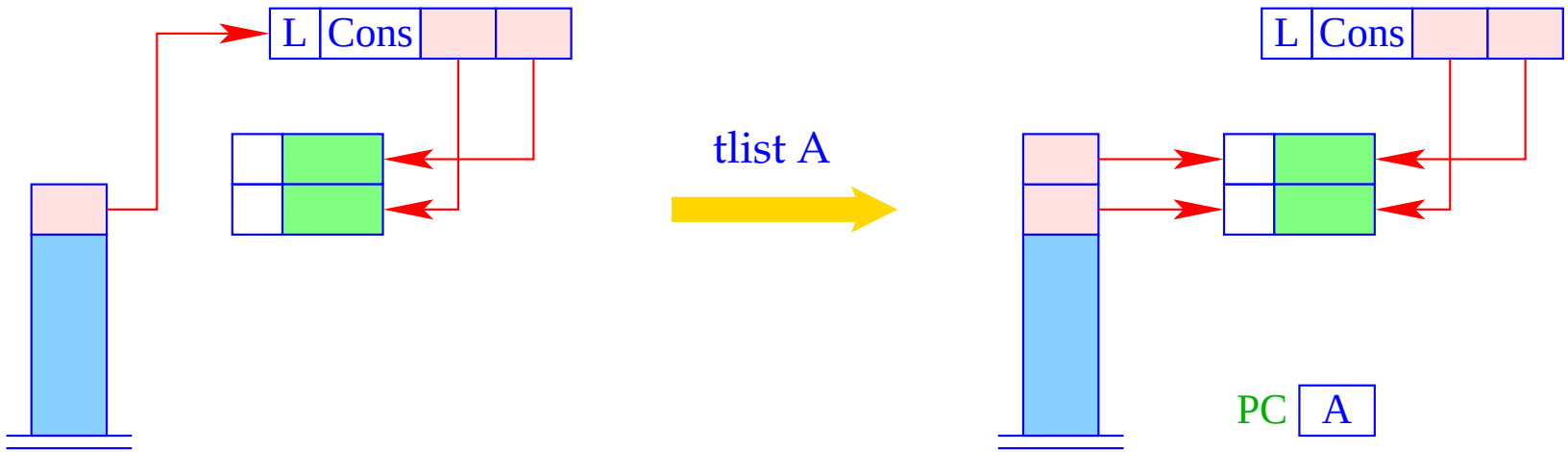
$$
\begin{array}{rcl}
\text{code}_V\ e\ \rho\ \text{sd} & = & \text{code}_V\ e_0\ \rho\ \text{sd} \\
 & & \text{tlist A} \\
 & & \text{code}_V\ e_1\ \rho\ \text{sd} \\
 & & \text{jump B} \\
 & \text{A :} & \text{code}_V\ e_2\ \rho'\ (\text{sd}+2) \\
 & & \text{slide 2} \\
 & \text{B :} & \ldots
\end{array}
$$

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

The new instruction tlist A does the necessary checks and (in the case of Cons) allocates two new local variables:




```
h = S[SP];
if (H[h] != (L,...)
    Error “no list!”;
if (H[h] == (_,Nil)) SP- -;
     ...
```




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

Example: The (disentangled) body of the function app with app $\mapsto (G, 0)$:

```
0        targ 2          3        pushglob 0      0    C:  mark D
0        pushloc 0       4        pushloc 2       3        pushglob 2
1        eval            5        pushloc 6       4        pushglob 1
1        tlist A         6        mkvec 3         5        pushglob 0
0        pushloc 1       4        mkclos C        6        eval
1        eval            4        cons            6        apply
1        jump B          0        slide 2         1    D:  update
2   A:   pushloc 1       3   B:   return 2
```

Note:

Datatypes with more than two constructors need a generalization of the tlist instruction, corresponding to a `switch`-instruction :-)

24.5 Closures of Tuples and Lists

The general schema for code_C can be optimized for tuples and lists:

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

25 Last Calls

A function application is called last call in an expression e if this application could deliver the value for e.

A last call usually is the outermost application of a defining expression.

A function definition is called tail recursive if all recursive calls are last calls.

Examples:

$r\ t\ (h : y)$ is a last call in $\quad$ **case** x **of** $[]\ \rightarrow y;\ \ h : t\ \rightarrow r\ t\ (h : y)$

$f\ (x - 1)$ is not a last call in $\quad$ **if** $x \leq 1$ **then** 1 **else** $x * f\ (x - 1)$

Observation:

Last calls in a function body need no new stack frame!

$\Longrightarrow$

Automatic transformation of tail recursion into loops!!!

The code for a last call $l \equiv (e'\ e_0 \ldots e_{m_1})$ inside a function f with k arguments must

1. allocate the arguments e_i and evaluate e' to a function (note: all this inside f's frame!);
2. deallocate the local variables and the k consumed arguments of f;
3. execute an apply.

$$
\begin{array}{lll}
\text{code}_V\ l\ \rho\ \text{sd} & = & \text{code}_C\ e_{m-1}\ \rho\ \text{sd} \\
 & & \text{code}_C\ e_{m-2}\ \rho\ (\text{sd}+1) \\
 & & \ldots \\
 & & \text{code}_C\ e_0\ \rho\ (\text{sd}+m-1) \\
 & & \text{code}_V\ e'\ \rho\ (\text{sd}+m) \qquad // \text{ Evaluation of the function} \\
 & & \text{move}\ r\ (m+1) \qquad // \text{ Deallocation of } r \text{ cells} \\
 & & \text{apply}
\end{array}
$$

where $r = sd + k$ is the number of stack cells to deallocate.

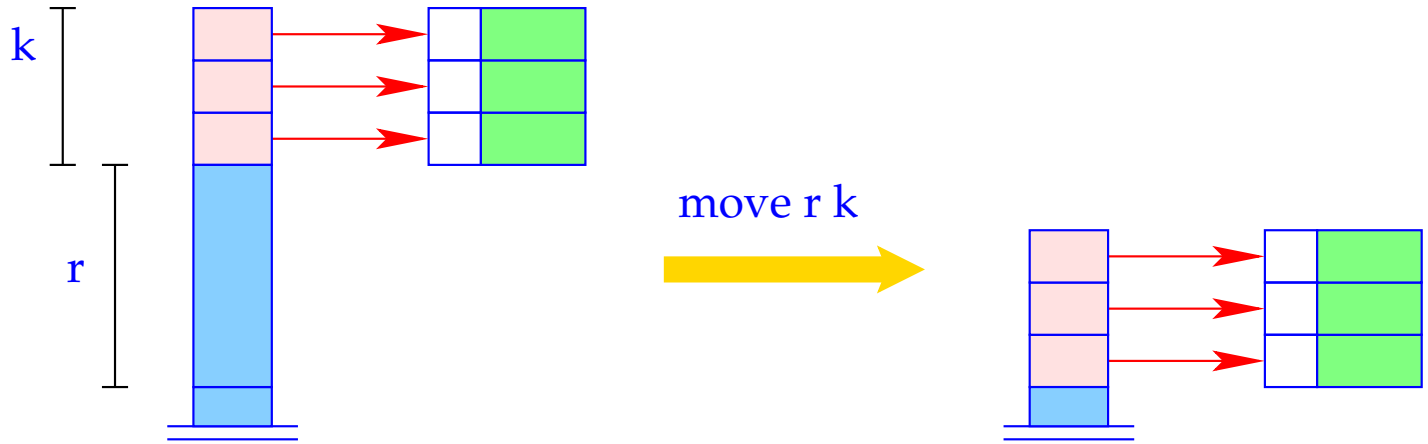
Example:

The body of the function

$$r = \mathbf{fn}\, x, y \;\Rightarrow\; \mathbf{case}\; x\; \mathbf{of}\; [] \;\rightarrow y;\; h : t \;\rightarrow r\; t\; (h : y)$$

```
0   targ 2          1          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                       slide 2
1   eval            3          pushloc 1       1   B:     return 2
```

Since the old stack frame is kept, return 2 will only be reached by the direct jump at the end of the []-alternative.




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

The Translation of Logic Languages

26 The Language Proll

Here, we just consider the core language Proll ("Prolog-light" :-). In particular, we omit:

- arithmetic;
- the cut operator;
- self-modification of programs through assert and retract.

Example:

$$
\begin{array}{lcl}
\mathsf{bigger}(X, Y) & \leftarrow & X = \mathit{elephant}, Y = \mathit{horse} \\
\mathsf{bigger}(X, Y) & \leftarrow & X = \mathit{horse}, Y = \mathit{donkey} \\
\mathsf{bigger}(X, Y) & \leftarrow & X = \mathit{donkey}, Y = \mathit{dog} \\
\mathsf{bigger}(X, Y) & \leftarrow & X = \mathit{donkey}, Y = \mathit{monkey} \\
\mathsf{is\_bigger}(X, Y) & \leftarrow & \mathsf{bigger}(X, Y) \\
\mathsf{is\_bigger}(X, Y) & \leftarrow & \mathsf{bigger}(X, Z), \mathsf{is\_bigger}(Z, Y) \\
? \quad \mathsf{is\_bigger}(\mathit{elephant}, \mathit{dog}) & &
\end{array}
$$

A More Realistic Example:

$$\begin{array}{lll}
\mathsf{app}(X, Y, Z) & \leftarrow & X = [\,],\ Y = Z \\
\mathsf{app}(X, Y, Z) & \leftarrow & X = [H|X'],\ Z = [H|Z'],\ \mathsf{app}(X', Y, Z') \\
? \quad \mathsf{app}(X, [Y, c], [a, b, Z]) & &
\end{array}$$

A More Realistic Example:

$$\begin{array}{lll} \mathsf{app}(X, Y, Z) & \leftarrow & X = [\,],\ Y = Z \\ \mathsf{app}(X, Y, Z) & \leftarrow & X = [H|X'],\ Z = [H|Z'],\ \mathsf{app}(X', Y, Z') \\ ? \quad \mathsf{app}(X, [Y, c], [a, b, Z]) & & \end{array}$$

Remark:

$[\,]$	$==$	the atom empty list
$[H\|Z]$	$==$	binary constructor application
$[a, b, Z]$	$==$	shortcut for: $[a\|[b\|[Z\|[\,]]]]$

A program p is constructed as follows:

$$
\begin{array}{lcl}
t & ::= & a \mid X \mid \_ \mid f(t_1, \ldots, t_n) \\
g & ::= & p(t_1, \ldots, t_k) \mid X = t \\
c & ::= & p(X_1, \ldots, X_k) \leftarrow g_1, \ldots, g_r \\
p & ::= & c_1 \ldots . c_m ? g
\end{array}
$$

- A term t either is an atom, a variable, an anonymous variable or a constructor application.
- A goal g either is a literal, i.e., a predicate call, or a unification.
- A clause c consists of a head $p(X_1, \ldots, X_k)$ with predicate name and list of formal parameters together with a body, i.e., a sequence of goals.
- A program consists of a sequence of clauses together with a single goal as query.

Procedural View of Proll programs:

goal	$==$	procedure call
predicate	$==$	procedure
body	$==$	definition
term	$==$	value
unification	$==$	basic computation step
binding of variables	$==$	side effect

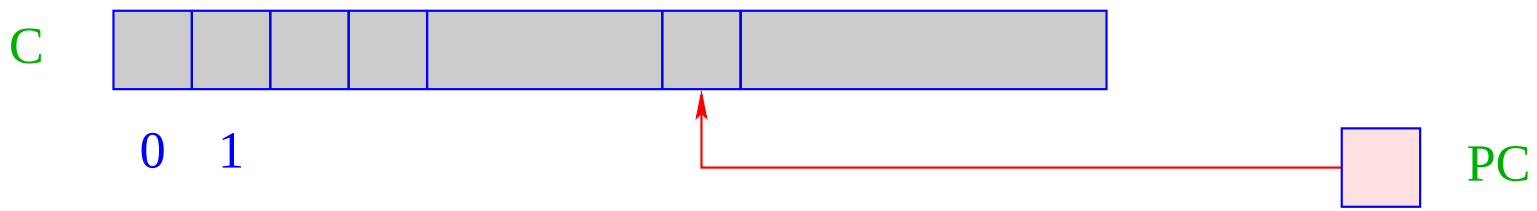
Note: Predicate calls ...

- ... do not have a return value.
- ... affect the caller through side effects only :-)
- ... may fail. Then the next definition is tried :-))

$\Longrightarrow$ backtracking

27 Architecture of the WiM:

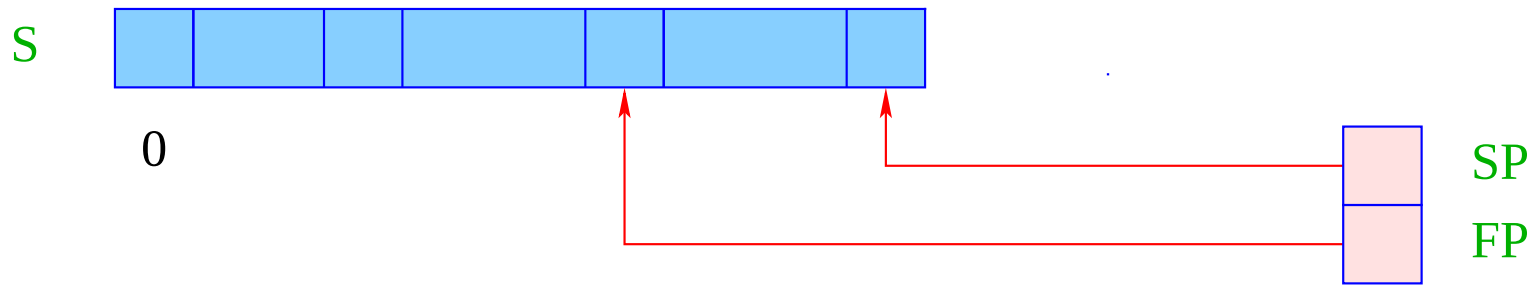
The Code Store:



C = Code store – contains WiM program;
every cell contains one instruction;

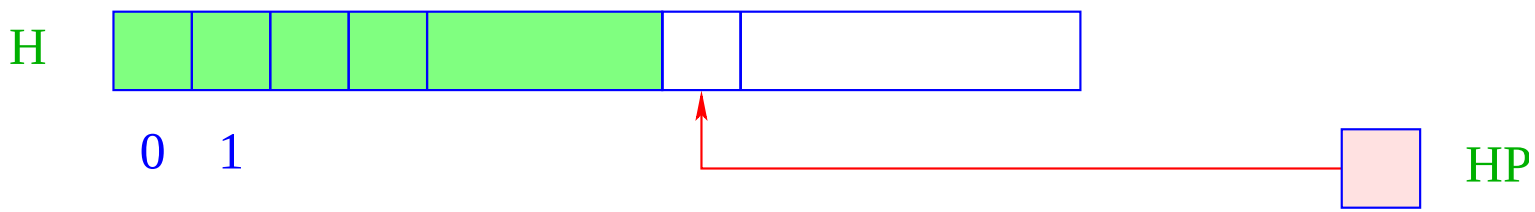
PC = Program Counter – points to the next instruction to executed;

The Runtime Stack:



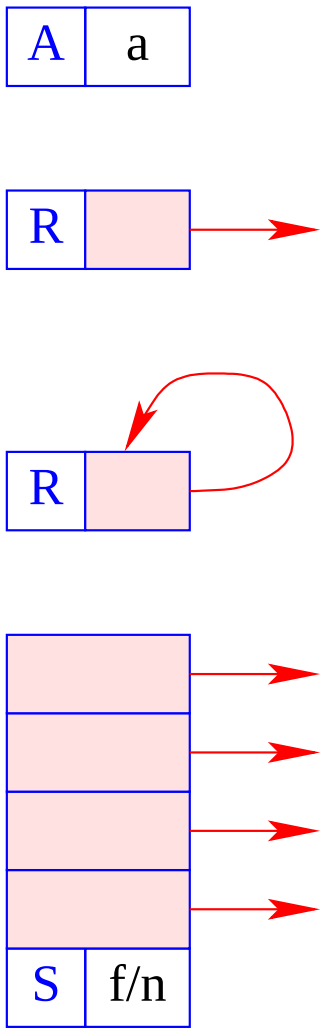
S	=	Runtime Stack – every cell may contain a value or an address;
SP	=	Stack Pointer – points to the topmost occupied cell;
FP	=	Frame Pointer – points to the current stack frame.
		Frames are created for predicate calls,
		contain cells for each variable of the current clause

The Heap:



H	=	Heap for dynamicly constructed terms;
HP	=	Heap-Pointer – points to the first free cell;

- The heap in maintained like a stack as well :-)
- A new-instruction allocates a object in H.
- Objects are tagged with their types (as in the MaMa) ...



atom — 1 cell

variable — 1 cell

unbound variable — 1 cell

structure — (n+1) cells

28 Construction of Terms in the Heap

Parameter terms of goals (calls) are constructed in the heap before passing.

Assume that the address environment ρ returns, for each clause variable X its address (relative to FP) on the stack. Then $\text{code}_A\ t\ \rho$ should ...

- construct (a presentation of) t in the heap; and
- return a reference to it on top of the stack.

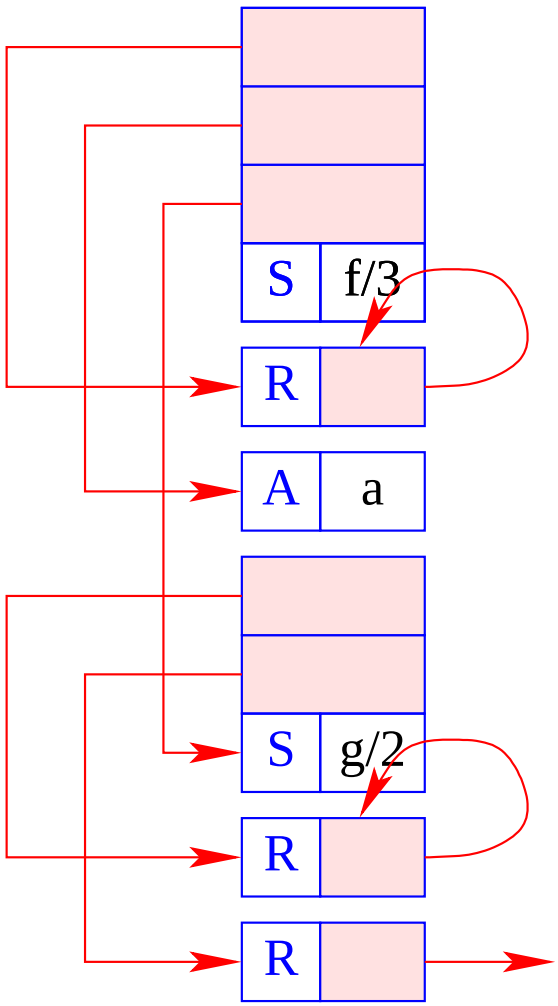
Idea:

- Construct the tree during a post-order traversal of t
- with one instruction for each new node!

Example: $t \equiv f(g(X, Y), a, Z)$.

Assume that X is initialized, i.e., $S[FP + \rho\, X]$ contains already a reference, Y and Z are not yet initialized.

Representing $t \equiv f(g(X,Y),a,Z)$:



reference to X

For a distinction, we mark occurrences of already initialized variables through over-lining (e.g. $\bar{X}$).

Note: Arguments are always initialized!

Then we define:

$$
\begin{array}{lcl}
\text{code}_A\ a\ \rho & = & \text{putatom a} \\
\text{code}_A\ X\ \rho & = & \text{putvar}\ (\rho\ X) \\
\text{code}_A\ \bar{X}\ \rho & = & \text{putref}\ (\rho\ X) \\
\text{code}_A\ \_\ \rho & = & \text{putanon}
\end{array}
\qquad
\begin{array}{lcl}
\text{code}_A\ f(t_1, \ldots, t_n)\ \rho & = & \text{code}_A\ t_1\ \rho \\
 & & \ldots \\
 & & \text{code}_A\ t_n\ \rho \\
 & & \text{putstruct f/n}
\end{array}
$$

For a distinction, we mark occurrences of already initialized variables through over-lining (e.g. $\bar{X}$).

Note: Arguments are always initialized!

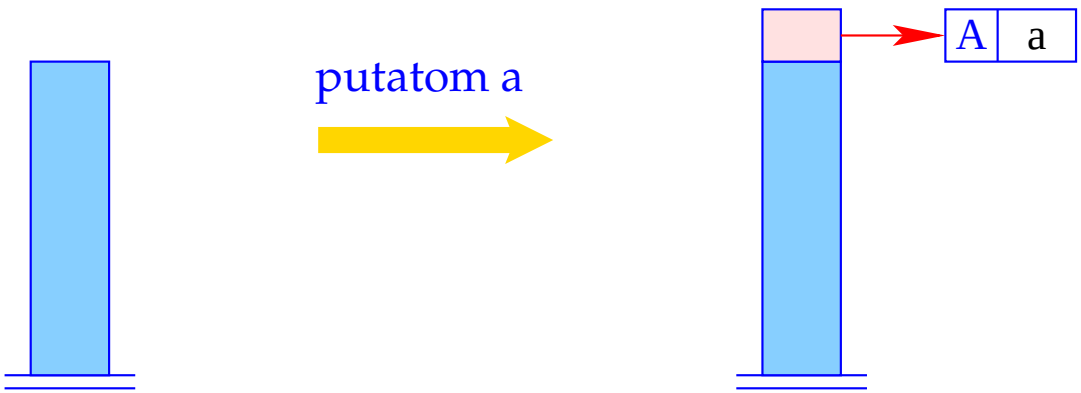
Then we define:

$$
\begin{array}{lcl}
\text{code}_A\ a\ \rho & = & \text{putatom a} \\
\text{code}_A\ X\ \rho & = & \text{putvar}\ (\rho\ X) \\
\text{code}_A\ \bar{X}\ \rho & = & \text{putref}\ (\rho\ X) \\
\text{code}_A\ \_\ \rho & = & \text{putanon}
\end{array}
\qquad
\begin{array}{lcl}
\text{code}_A\ f(t_1, \ldots, t_n)\ \rho & = & \text{code}_A\ t_1\ \rho \\
 & & \ldots \\
 & & \text{code}_A\ t_n\ \rho \\
 & & \text{putstruct f/n}
\end{array}
$$

For $f(g(X, Y), a, Z)$ and $\rho = \{X \mapsto 1, Y \mapsto 2, Z \mapsto 3\}$ this results in the sequence:

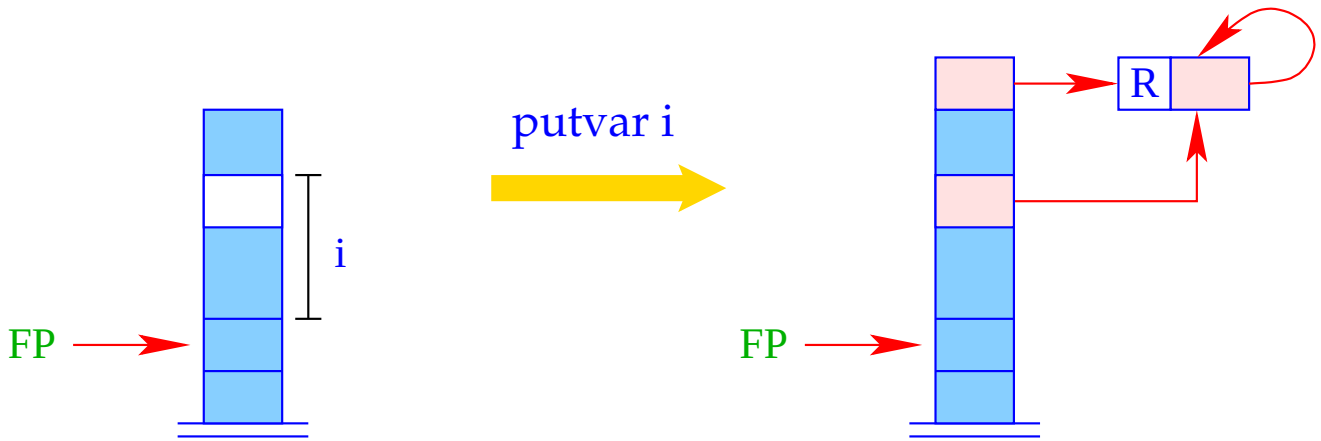
putref 1	putatom a
putvar 2	putvar 3
putstruct g/2	putstruct f/3

The instruction putatom a constructs an atom in the heap:



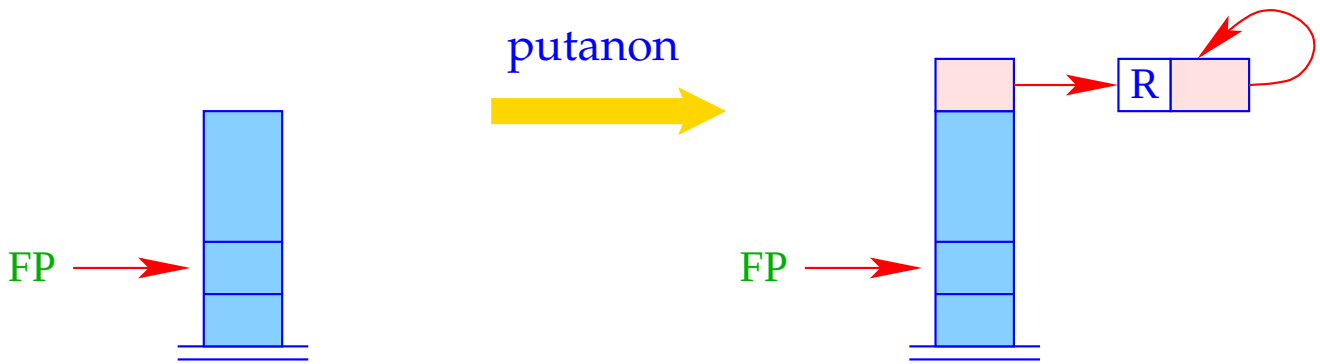
SP++; S[SP] = new (A,a);

The instruction putvar i introduces a new unbound variable and additionally initializes the corresponding cell in the stack frame:



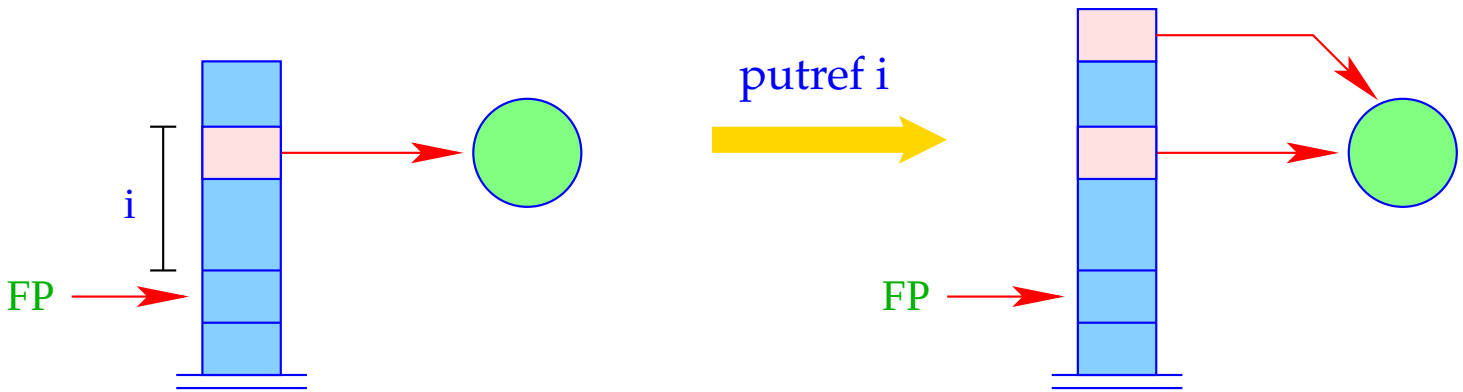

```
SP = SP + 1;
S[SP] = new (R, HP);
S[FP + i] = S[SP];
```

The instruction putanon introduces a new unbound variable but does not store a reference to it in the stack frame:



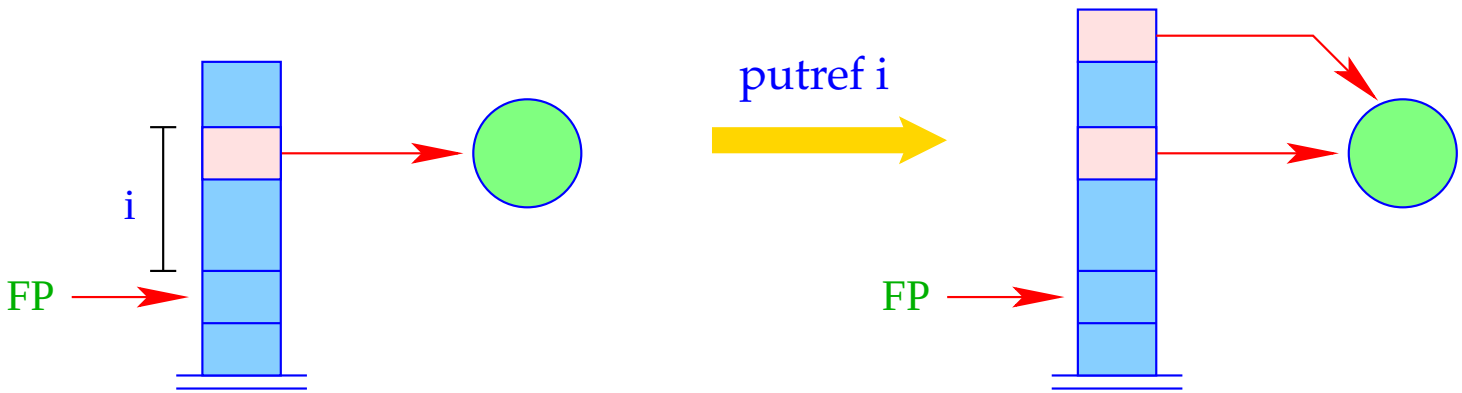
SP = SP + 1;
S[SP] = new (R, HP);

The instruction putref i pushes the value of the variable onto the stack:



SP = SP + 1;
S[SP] = deref S[FP + i];

The instruction putref i pushes the value of the variable onto the stack:

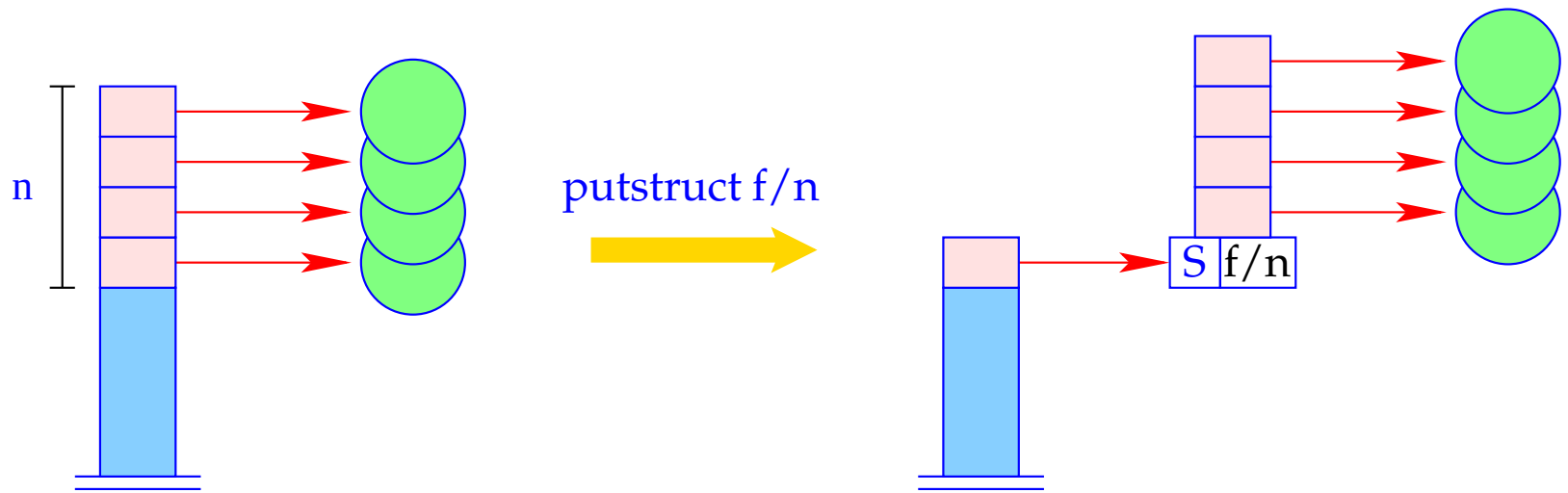


SP = SP + 1;
S[SP] = deref S[FP + i];

The auxiliary function deref contracts chains of references:

```
ref deref (ref v) {
      if (H[v]==(R,w) && v!=w) return deref (w);
      else return v;
}
```

The instruction putstruct i builds a constructor application in the heap:




```
v = new (S, f, n);
SP = SP - n + 1;
for (i=1; i<=n; i++)
    H[v + i] = S[SP + i -1];
S[SP] = v;
```

Remarks:

- The instruction putref i does not just push the reference from $S[FP + i]$ onto the stack, but also dereferences it as much as possible

  $\Longrightarrow$ maximal contraction of reference chains.

- In constructed terms, references always point to smaller heap addresses.

  Also otherwise, this will be often the case. Sadly enough, it cannot be guaranteed in general :-(

29 The Translation of Literals (Goals)

Idea:

- Literals are treated as procedure calls.
- We first allocate a stack frame.
- Then we construct the actual parameters (in the heap)
- ... and store references to these into the stack frame.
- Finally, we jump to the code for the procedure/predicate.

$$
\begin{array}{lcll}
\text{code}_G\ p(t_1, \ldots, t_k)\ \rho & = & \text{mark B} & \text{// allocates the stack frame} \\
 & & \text{code}_A\ t_1\ \rho & \\
 & & \ldots & \\
 & & \text{code}_A\ t_k\ \rho & \\
 & & \text{call p/k} & \text{// calls the procedure p/k} \\
 & \text{B :} & \ldots &
\end{array}
$$

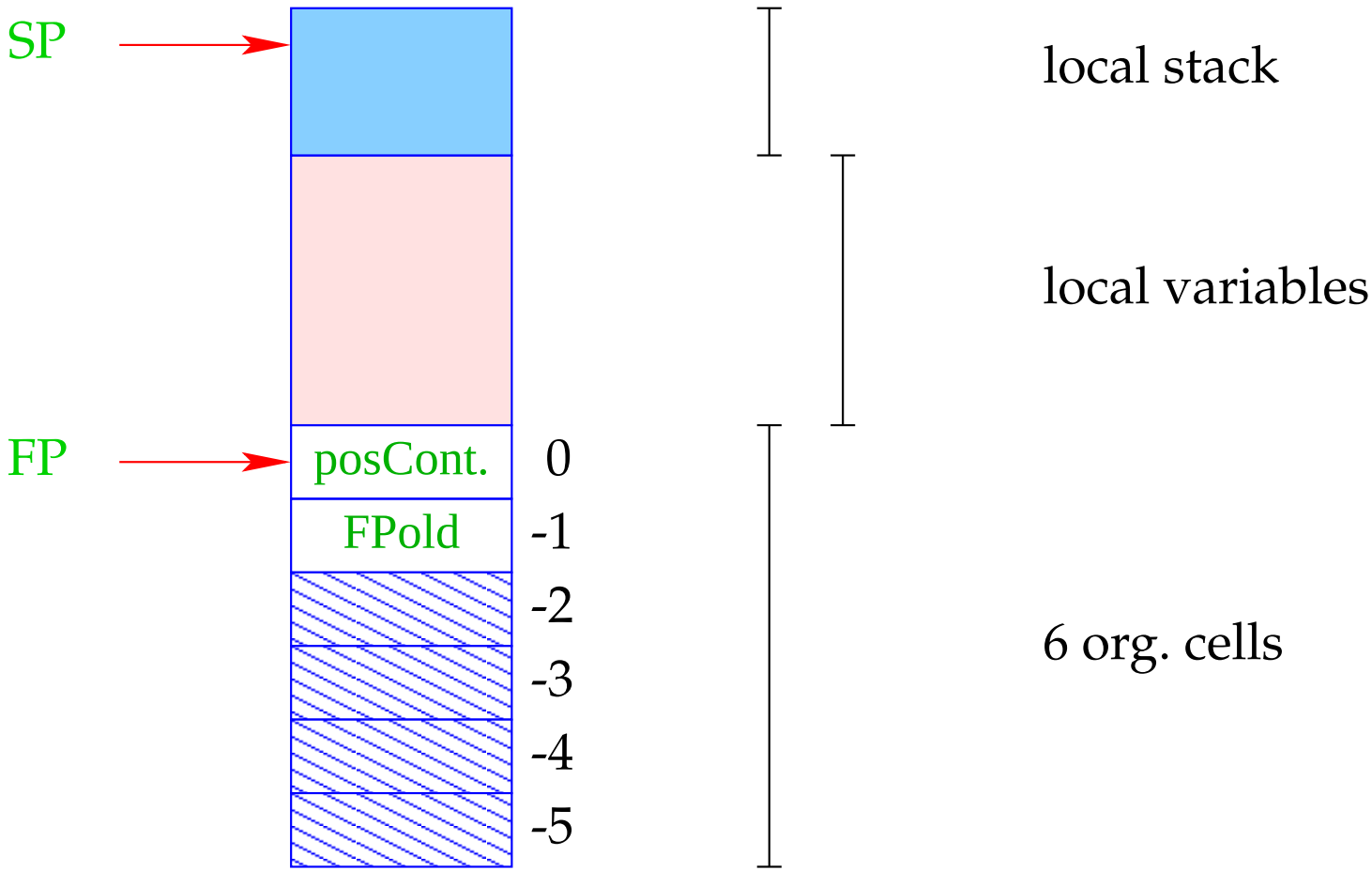
$\text{code}_G\ p(t_1, \ldots, t_k)\ \rho$ = mark B // allocates the stack frame
$\text{code}_A\ t_1\ \rho$
...
$\text{code}_A\ t_k\ \rho$
call p/k // calls the procedure p/k
B : ...

Example: $p(a, X, g(\bar{X}, Y))$ with $\rho = \{X \mapsto 1, Y \mapsto 2\}$

We obtain:

```
mark B          putref 1              call p/3
putatom a       putvar 2        B:    ...
putvar 1        putstruct g/2
```

Stack Frame of the WiM:

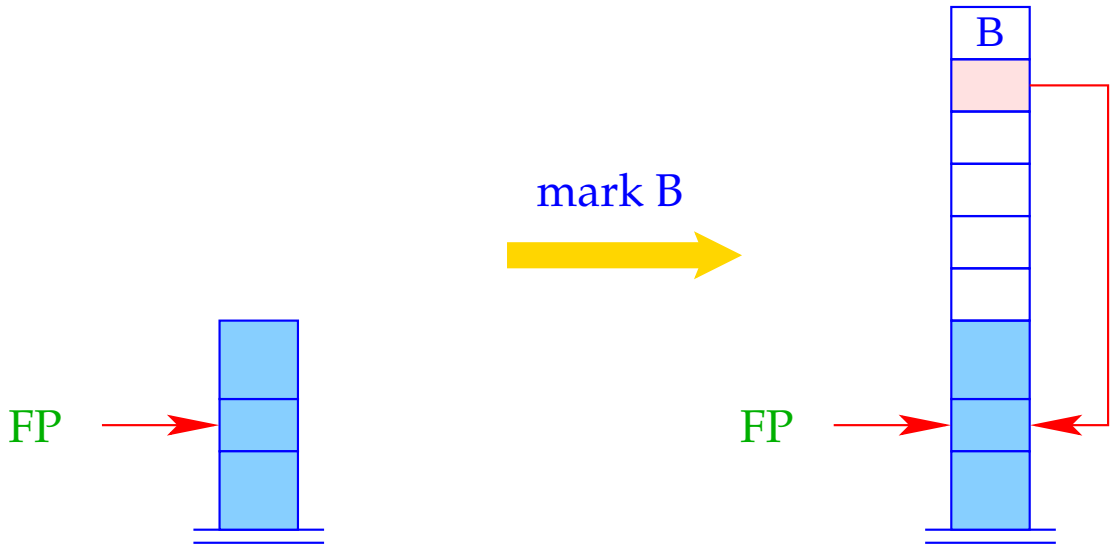


Remarks:

- The positive continuation address records where to continue after successful treatment of the goal.
- Additional organizational cells are needed for the implementation of backtracking

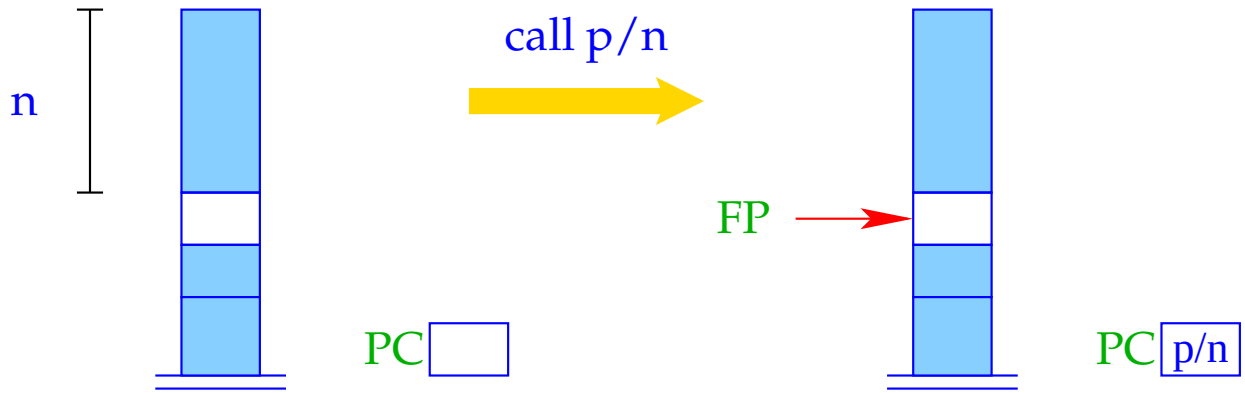
  $\Longrightarrow$ will be discussed at the translation of predicates.

The instruction mark B allocates a new stack frame:



SP = SP + 6;
S[SP] = B; S[SP-1] = FP;

The instruction call p/n calls the n-ary predicate p :



FP = SP - n;
PC = p/n;

30 Unification

Convention:

- By $\tilde{X}$, we denote an occurrence of X;
  it will be translated differently depending on whether the variable is initialized or not.
- We introduce the macro $\text{put}\ \tilde{X}\ \rho$:

$$\begin{aligned} \text{put}\ X\ \rho &= \text{putvar}\ (\rho\, X) \\ \text{put}\ \_\ \rho &= \text{putanon} \\ \text{put}\ \bar{X}\ \rho &= \text{putref}\ (\rho\, X) \end{aligned}$$

Let us translate the unification $\tilde{X} = t$.

Idea 1:

- Push a reference to (the binding of) X onto the stack;
- Construct the term t in the heap;
- Invent a new instruction implementing the unification :-)

Let us translate the unification $\tilde{X} = t$.

Idea 1:

- Push a reference to (the binding of) X onto the stack;
- Construct the term t in the heap;
- Invent a new instruction implementing the unification :-)

$$\begin{aligned} \text{code}_G\ (\tilde{X} = t)\ \rho \quad = \quad & \text{put}\ \tilde{X}\ \rho \\ & \text{code}_A\ t\ \rho \\ & \text{unify} \end{aligned}$$

Example:

Consider the equation:

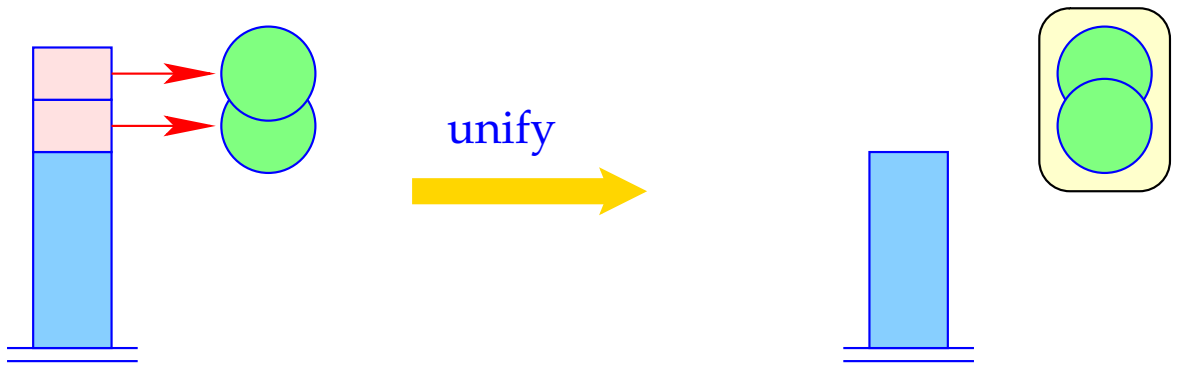
$$\bar{U} = f(g(\bar{X}, Y), a, Z)$$

Then we obtain for an address environment

$$\rho = \{X \mapsto 1, Y \mapsto 2, Z \mapsto 3, U \mapsto 4\}$$

putref 4	putref 1	putatom a	unify
	putvar 2	putvar 3	
	putstruct g/2	putstruct f/3	

The instruction unify calls the run-time function `unify()` for the topmost two references:




```
unify (S[SP-1], S[SP]);
SP = SP–2;
```

The Function `unify()`

- ... takes two heap addresses.

  For each call, we guarantee that these are maximally de-referenced.

- ... checks whether the two addresses are already identical.

  If so, does nothing :-)

- ... binds younger variables (larger addresses) to older variables (smaller addresses);

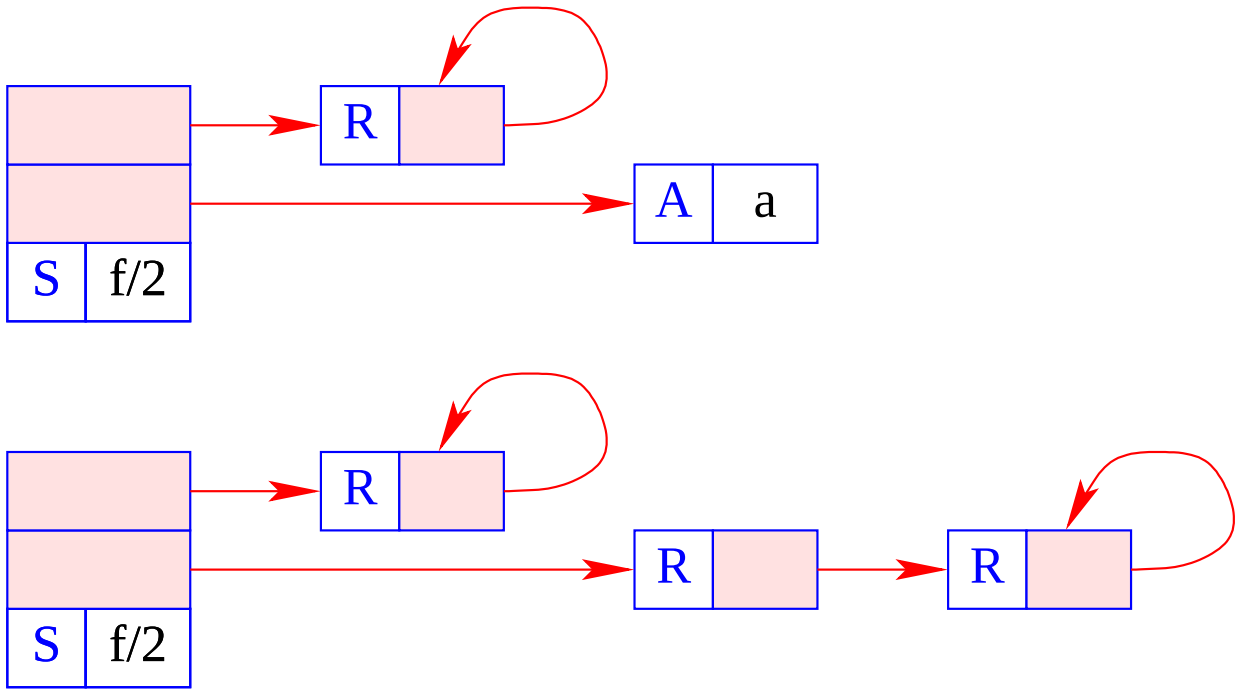
- ... when binding a variable to a term, checks whether the variable occurs inside the term $\Longrightarrow$ occur-check;

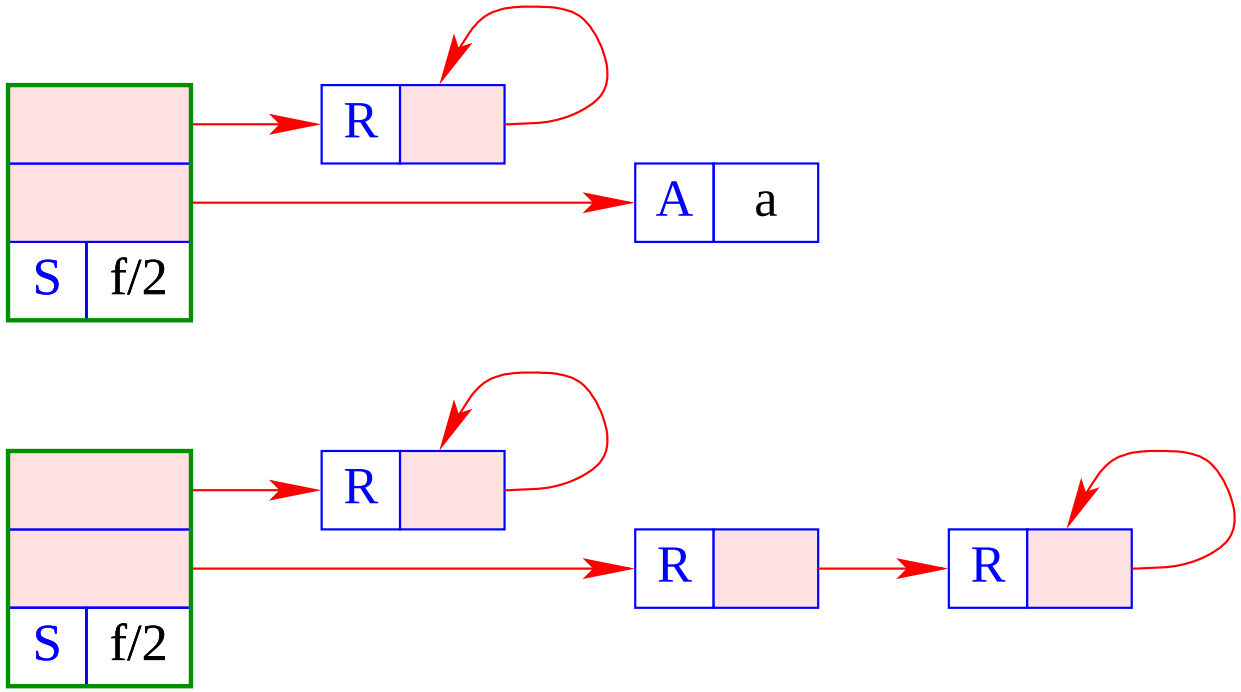
- ... records newly created bindings;

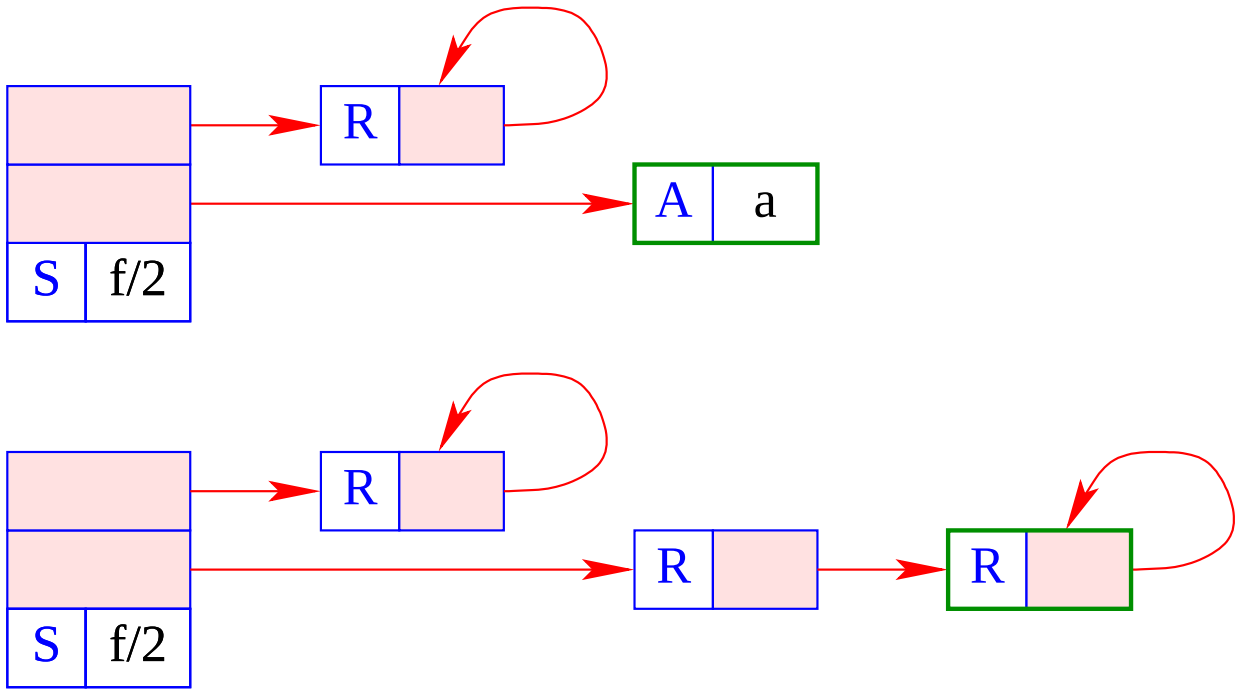
- ... may fail. Then backtracking is initiated.

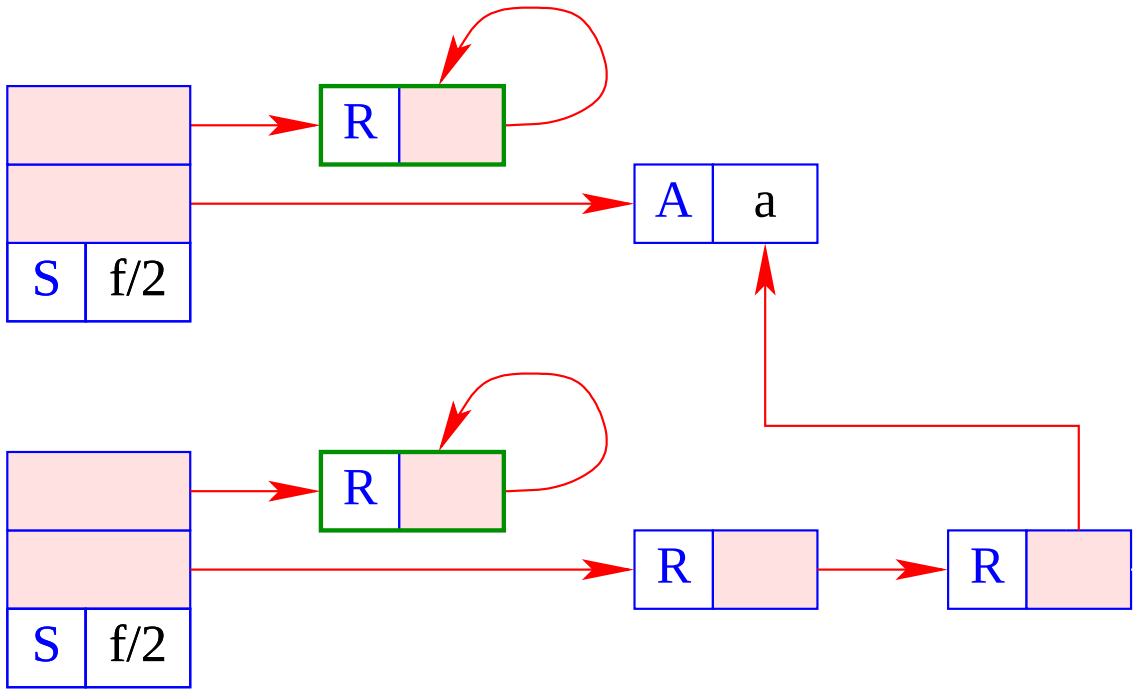
```
bool unify (ref u, ref v) {
   if (u == v) return true;
   if (H[u] == (R,_)) {
      if (H[v] == (R,_)) {
         if (u>v) {
            H[u] = (R,v); trail (u); return true;
         } else {
            H[v] = (R,u); trail (v); return true;
         }
      } elseif (check (u,v)) {
         H[u] = (R,v); trail (u); return true;
      } else {
         backtrack(); return false;
      }
   }
   ...
```

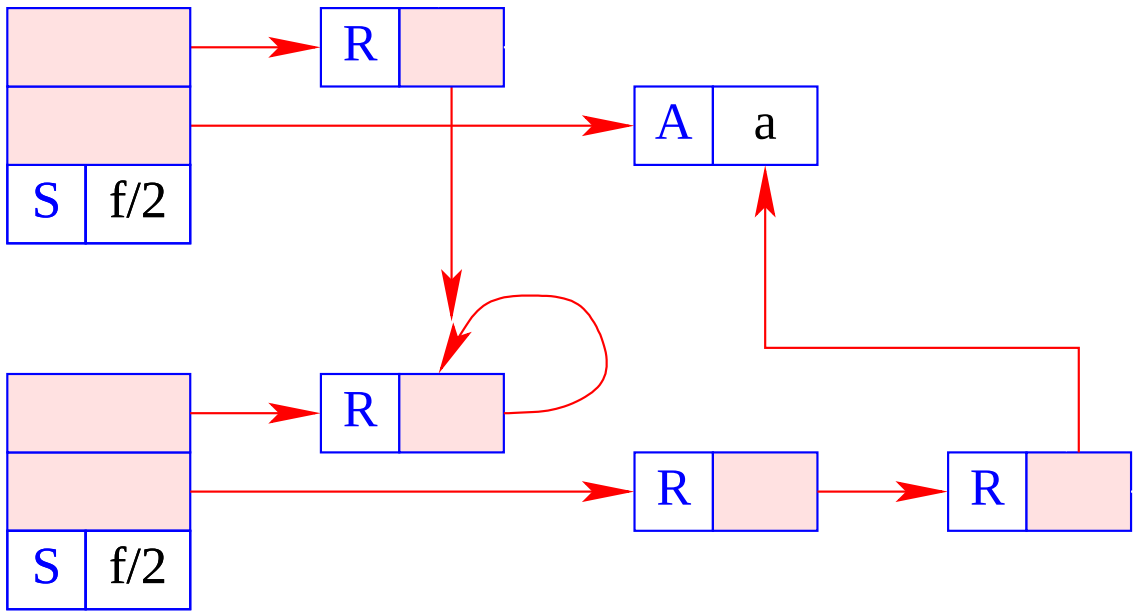
```
...
 if ((H[v] == (R,_)) {
      if (check (v,u)) {
         H[v] = (R,u); trail (v); return true;
      } else {
         backtrack(); return false;
      }
   }
   if (H[u]==(A,a) && H[v]==(A,a))
      return true;
   if (H[u]==(S, f/n) && H[v]==(S, f/n)) {
      for (int i=1; i<=n; i++)
         if(!unify (deref (H[u+i]), deref (H[v+i])) return false;
      return true;
   }
   backtrack(); return false;
}
```


R
A
a
S
f/2
R
R
R
S
f/2


R
A
a
S
f/2
R
R
R
S
f/2


R
A
a
S
f/2
R
R
R
S
f/2


R
A
a
S
f/2
R
R
R
S
f/2


R
A
a
S
f/2
R
R
R
S
f/2

- The run-time function `trail()` records the a potential new binding.
- The run-time function `backtrack()` initiates backtracking.
- The auxiliary function `check()` performs the occur-check: it tests whether a variable (the first argument) occurs inside a term (the second argument).
- Often, this check is skipped, i.e.,

```
bool check (ref u, ref v) { return true;}
```

Otherwise, we could implement the run-time function `check()` as follows:

```
bool check (ref u, ref v) {
   if (u == v) return false;
   if (H[v] == (S, f/n)) {
      for (int i=1; i<=n; i++)
         if (!check(u, deref (H[v+i])))
            return false;
   return true;
}
```

Discussion:

- The translation of an equation $\tilde{X} = t$ is very simple :-)
- Often the constructed cells immediately become garbage :-(

Idea 2:

- Push a reference to the run-time binding of the left-hand side onto the stack.
- Avoid to construct sub-terms of t whenever possible !
- Translate each node of t into an instruction which performs the unifcation with this node !!

Discussion:

- The translation of an equation $\tilde{X} = t$ is very simple :-)
- Often the constructed cells immediately become garbage :-(

Idea 2:

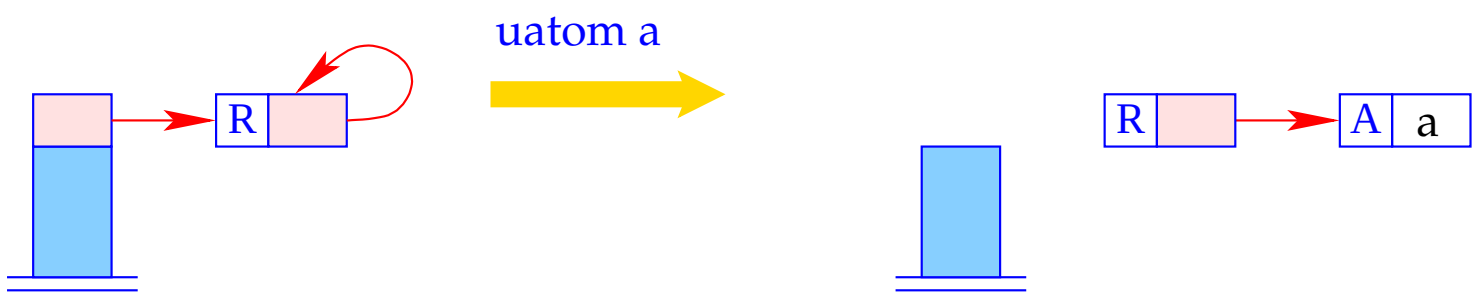
- Push a reference to the run-time binding of the left-hand side onto the stack.
- Avoid to construct sub-terms of t whenever possible !
- Translate each node of t into an instruction which performs the unifcation with this node !!

$$\begin{aligned} \text{code}_G\ (\tilde{X} = t)\ \rho \quad &= \quad \text{put}\ \tilde{X}\ \rho \\ &\qquad\quad \text{code}_U\ t\ \rho \end{aligned}$$

Let us first consider the unifcation code for atoms and variables only:

$$
\begin{array}{lcll}
\text{code}_U\ a\ \rho & = & \text{uatom a} & \\
\text{code}_U\ X\ \rho & = & \text{uvar}\ (\rho\, X) & \\
\text{code}_U\ \_\ \rho & = & \text{pop} & \\
\text{code}_U\ \bar{X}\ \rho & = & \text{uref}\ (\rho\, X) & \\
 & \ldots & & \text{// to be continued} \quad \text{:-)}
\end{array}
$$

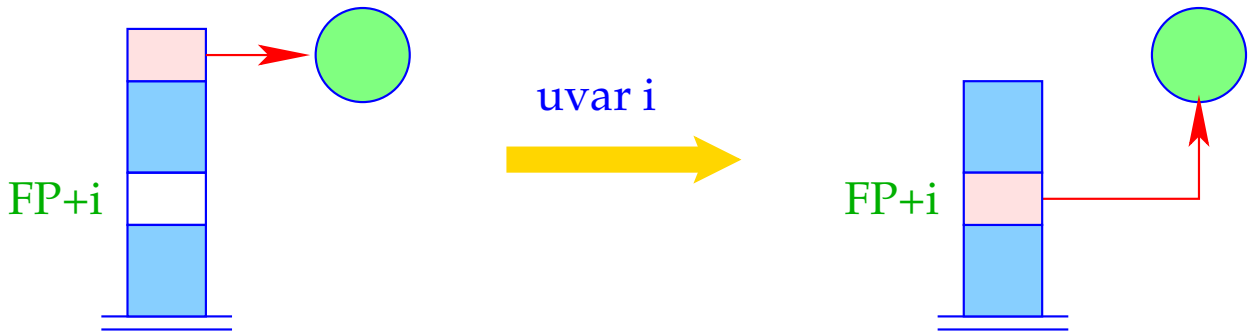
The instruction uatom a implements the unification with the atom a:




```
v = S[SP]; SP−−;
switch (H[v]) {
case (A, a):      break;
case (R,_) :      H[v] = (R, new (A, a));
                  trail (v); break;
default:          backtrack();
}
```

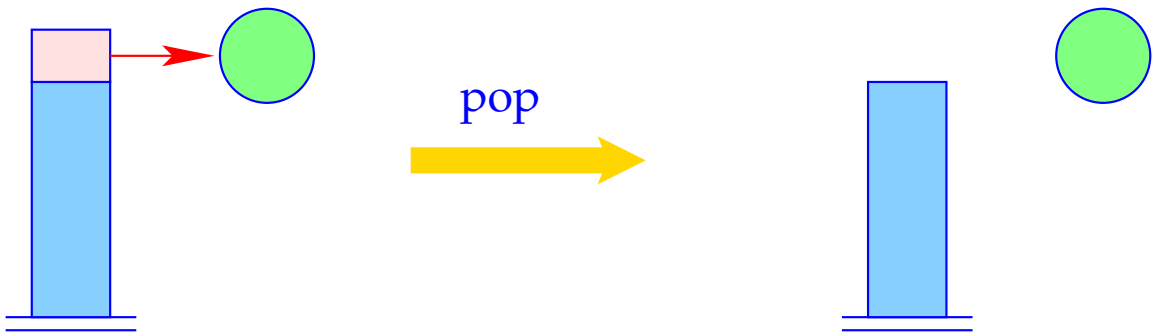
- The run-time function `trail()` records the a potential new binding.
- The run-time function `backtrack()` initiates backtracking.

The instruction uvar i implements the unification with an un-initialized variable:



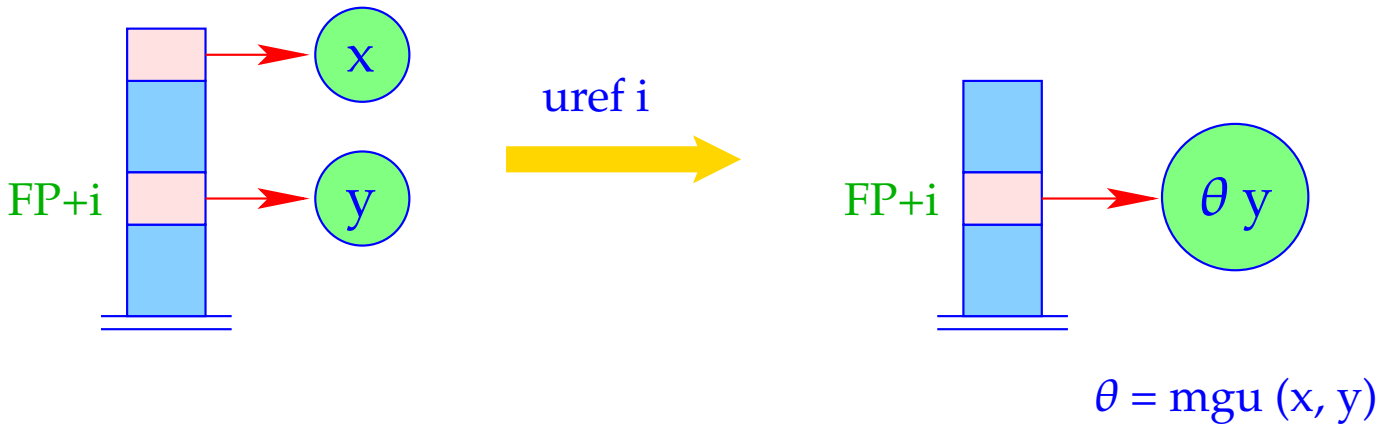
S[FP+i] = S[SP]; SP– –;

The instruction pop implements the unification with an anonymous variable. It always succeeds :-)



SP– –;

The instruction uref i implements the unification with an initialized variable:




```
unify (S[SP], deref (S[FP+i]));
SP– –;
```

It is only here that the run-time function `unify()` is called :-)

- The unification code performs a pre-order traversal over t.
- In case, execution hits at an unbound variable, we switch from checking to building :-)

$$
\begin{array}{lll}
\text{code}_U\ f(t_1,\ldots,t_n)\ \rho \quad = & \text{ustruct f/n } A & \text{// test} \\
& \text{son 1} & \\
& \text{code}_U\ t_1\ \rho & \\
& \ldots & \\
& \text{son n} & \\
& \text{code}_U\ t_n\ \rho & \\
& \text{up B} & \\
A: & \text{check } \mathit{ivars}(f(t_1,\ldots,t_n))\ \rho & \text{// occur-check} \\
& \text{code}_A\ f(t_1,\ldots,t_n)\ \rho & \text{// building !!} \\
& \text{bind} & \text{// creation of bindings} \\
B: & \ldots &
\end{array}
$$

The Building Block:

Before constructing the new (sub-) term t' for the binding, we must exclude that it contains the variable X' on top of the stack !!!

This is the case iff the binding of no variable inside t' contains (a reference to) X'.

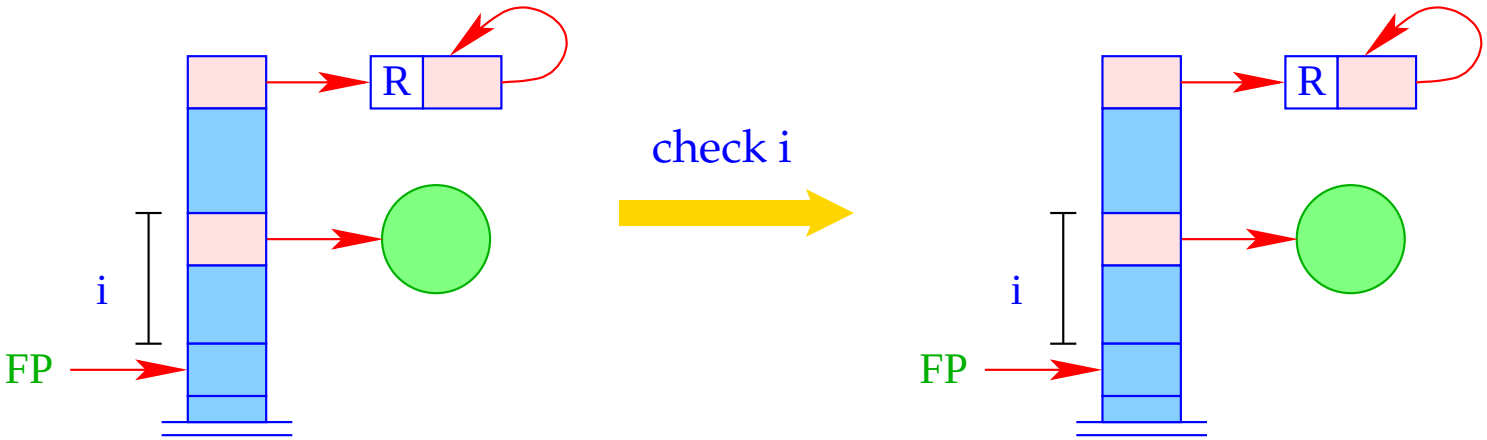
$\Longrightarrow$ $ivars(t')$ returns the set of already initialized variables of t.

$\Longrightarrow$ The macro check $\{Y_1, \ldots, Y_d\}\ \rho$ generates the necessary tests on the variables $Y_1, \ldots, Y_d$:

$$\begin{aligned} \text{check } \{Y_1, \ldots, Y_d\}\ \rho \quad = \quad & \text{check } (\rho\ Y_1) \\ & \text{check } (\rho\ Y_2) \\ & \ldots \\ & \text{check } (\rho\ Y_d) \end{aligned}$$

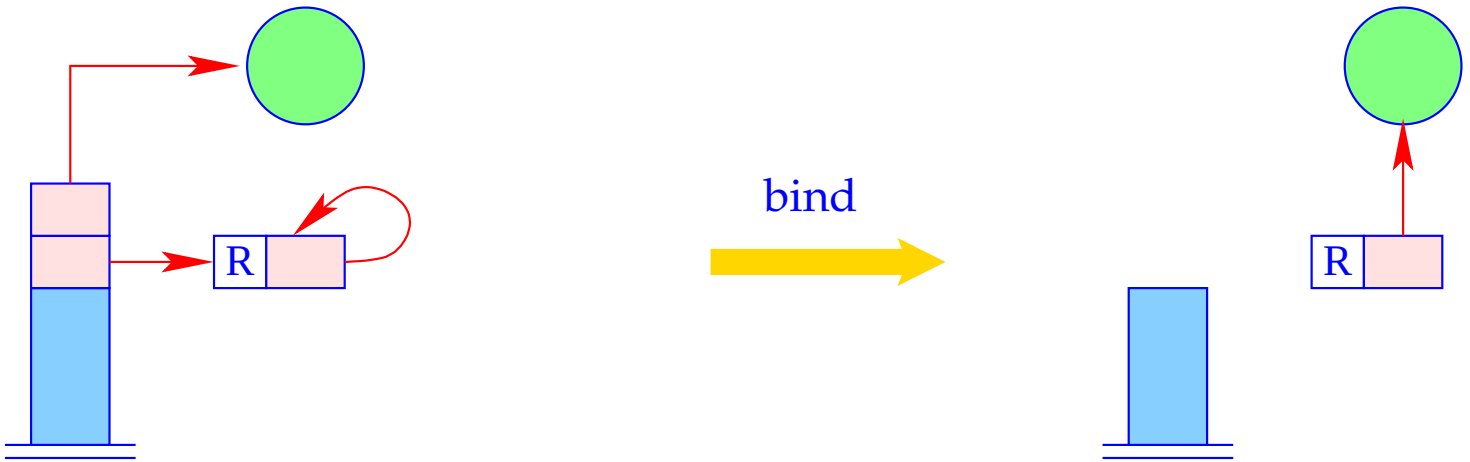
The instruction check i checks whether the (unbound) variable on top of the stack occurs inside the term bound to variable i.

If so, unification fails and backtracking is caused:




```
if (!check (S[SP], deref S[FP+i]))
    backtrack();
```

The instruction bind terminates the building block. It binds the (unbound) variable to the constructed term:




```
H[S[SP-1]] = (R, S[SP]);
trail (S[SP-1]);
SP = SP – 2;
```

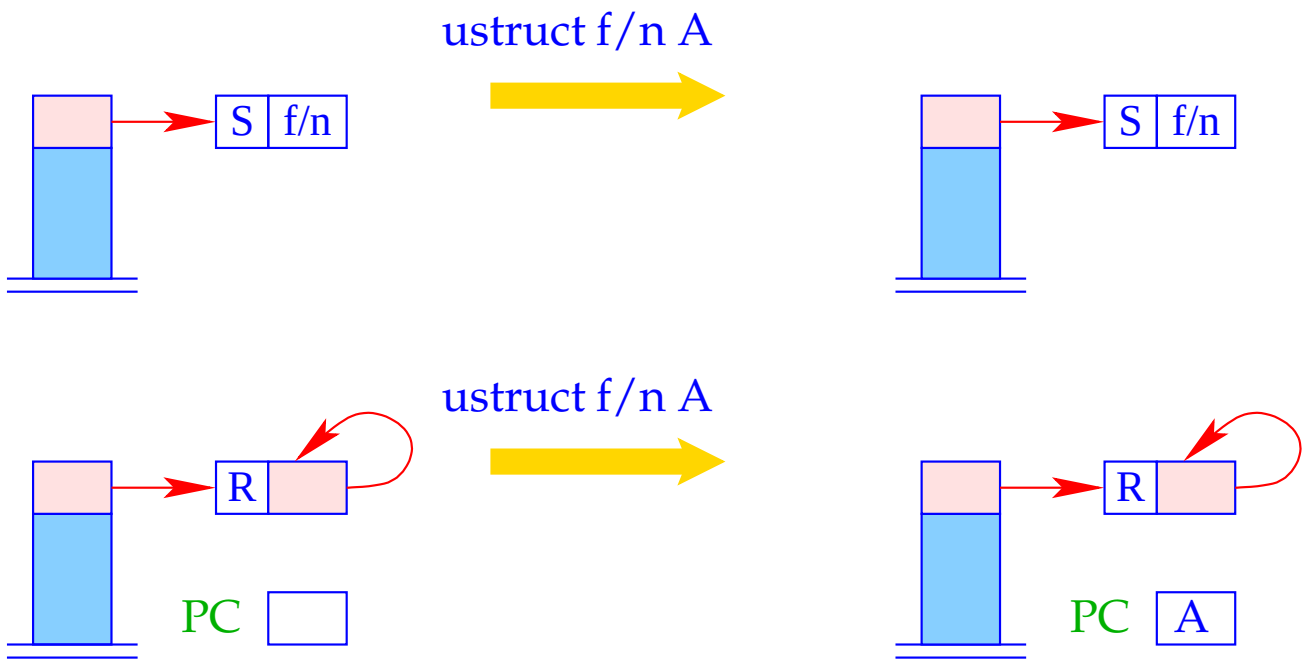
The Pre-Order Traversal:

- First, we test whether the topmost reference is an unbound variable.
  If so, we jump to the building block.
- Then we compare the root node with the constructor f/n.
- Then we recursively descend to the children.
- Then we pop the stack and proceed behind the unification code:

Once again the unification code for constructed terms:

$$
\begin{array}{lll}
\text{code}_U\ f(t_1,\ldots,t_n)\ \rho \quad = & \text{ustruct f/n } A & \text{// test} \\
& \text{son 1} & \text{// recursive descent} \\
& \text{code}_U\ t_1\ \rho & \\
& \ldots & \\
& \text{son n} & \text{// recursive descent} \\
& \text{code}_U\ t_n\ \rho & \\
& \text{up B} & \text{// ascent to father} \\
A: & \text{check } \mathit{ivars}(f(t_1,\ldots,t_n))\ \rho & \\
& \text{code}_A\ f(t_1,\ldots,t_n)\ \rho & \\
& \text{bind} & \\
B: & \ldots &
\end{array}
$$

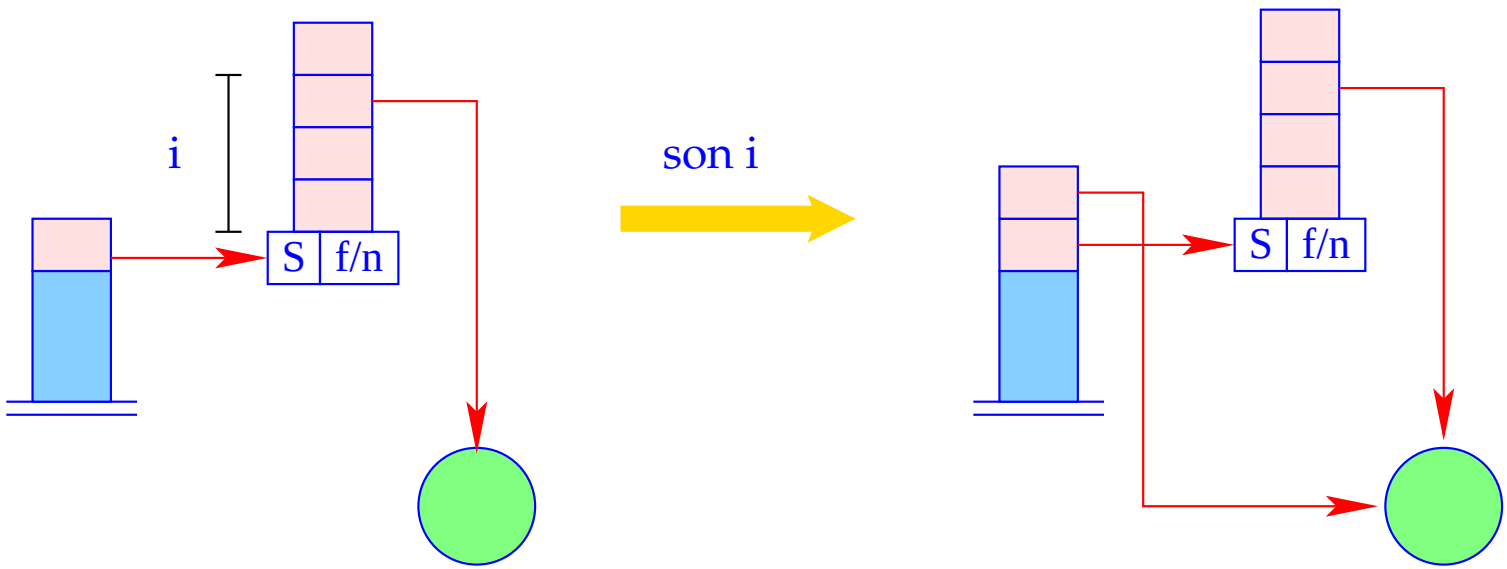
The instruction ustruct i implements the test of the root node of a structure:




```
switch (H[S[SP]]) {
case (S, f/n):    break;
case (R,_) :      PC = A; break;
default:          backtrack();
}
```

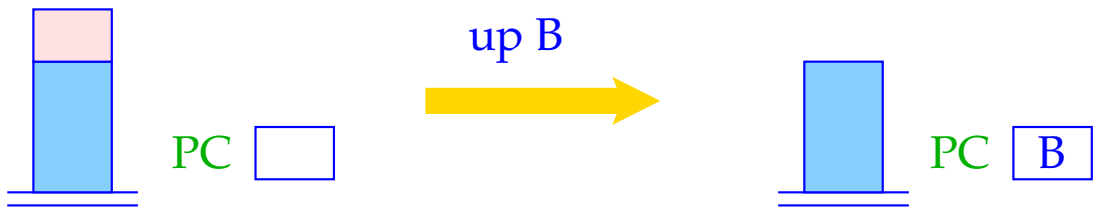
... the argument reference is not yet popped :-)

The instruction son i pushes the (reference to the) i-th sub-term from the structure pointed at from the topmost reference:



S[SP+1] = deref (H[S[SP]+i]); SP++;

It is the instruction up B which finally pops the reference to the structure:



SP– –; PC = B;

The continuation address B is the next address after the build-section.

Example:

For our example term $f(g(\bar{X}, Y), a, Z)$ and $\rho = \{X \mapsto 1, Y \mapsto 2, Z \mapsto 3\}$ we obtain:

```
ustruct f/3 A1          up B2          B2:  son 2          putvar 2
son 1                                       uatom a         putstruct g/2
ustruct g/2 A2     A2:  check 1             son 3           putatom a
son 1                   putref 1            uvar 3          putvar 3
uref 1                  putvar 2            up B1           putstruct f/3
son 2                   putstruct g/2  A1:   check 1         bind
uvar 2                  bind                putref 1   B1:  ...
```

Code size can grow quite considerably — for deep terms. In practice, though, deep terms are "rare" :-)

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 :-)

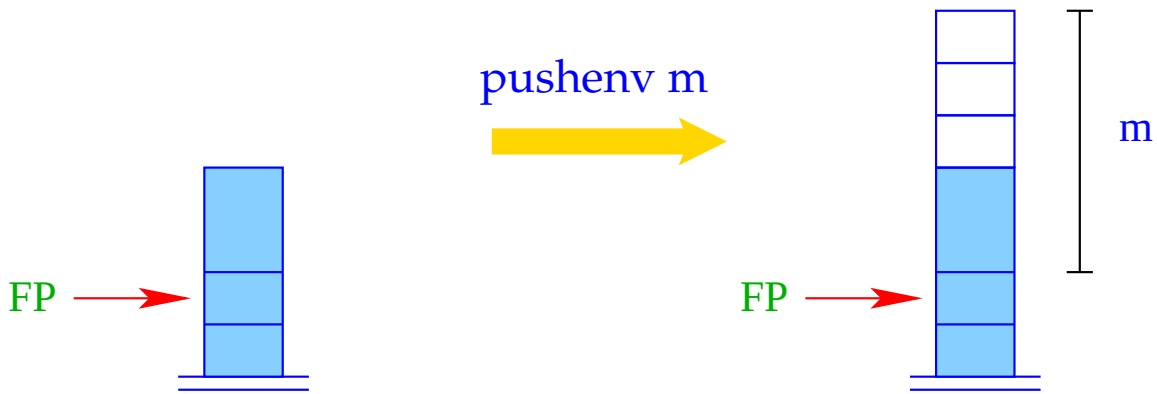
Then we translate:

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

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 $\quad f = 1 \quad$:

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

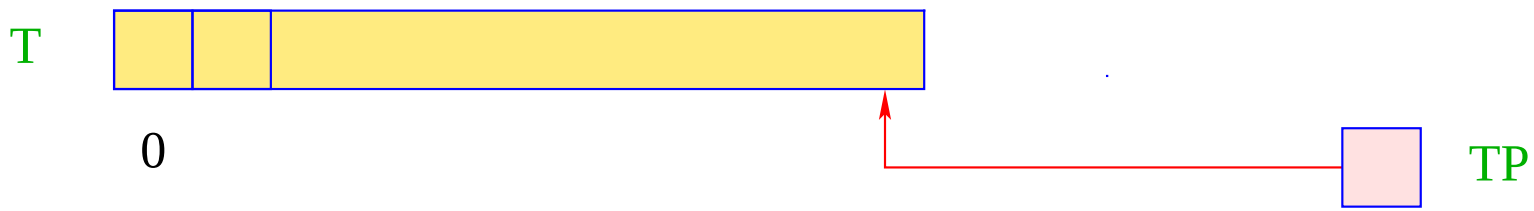
If q/k is defined through several clauses, the first alternative must be tried.

On failure, the next alternative must be tried

$$\Longrightarrow \qquad \text{backtracking} \quad \text{:-)}$$

32.1 Backtracking

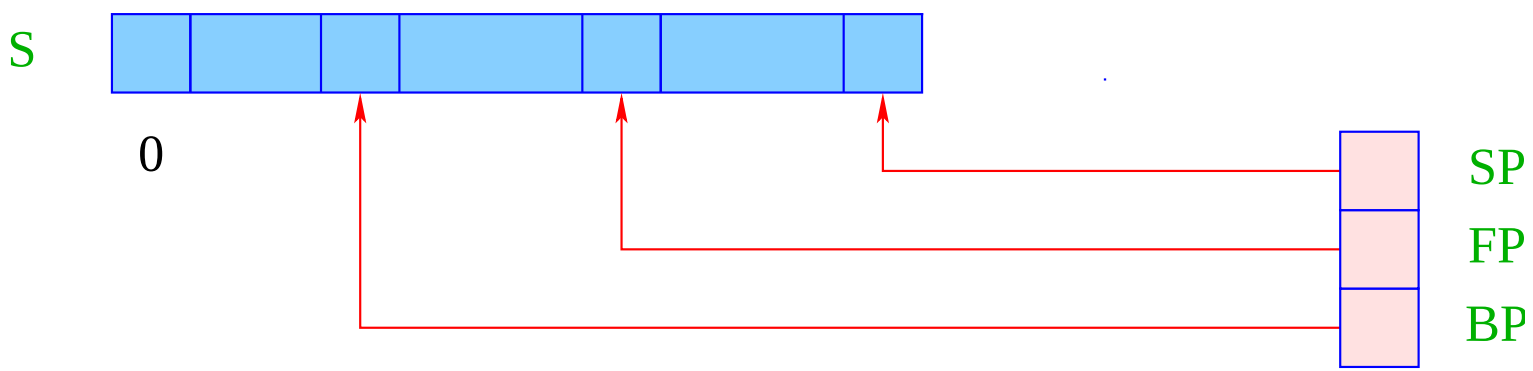
- Whenever unifcation fails, we call the run-time function `backtrack()`.
- The goal is to roll back the whole computation to the (dynamically :-) latest goal where another clause can be chosen $\Longrightarrow$ the last backtrack point.
- In order to undo intermediate variable bindings, we always have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

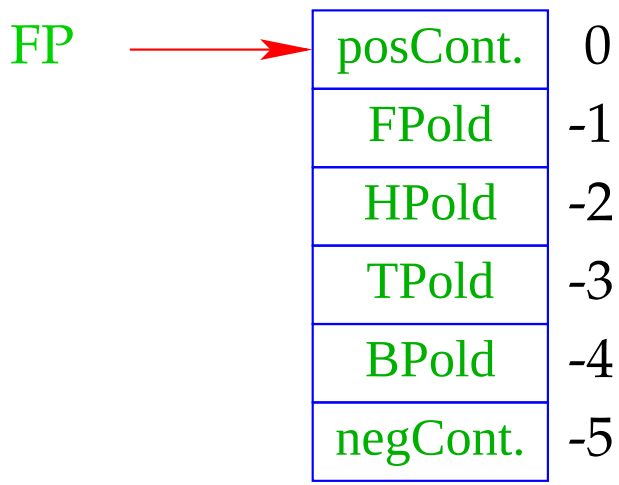
The current stack frame where backtracking should return to is pointed at by the extra register BP:



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:

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

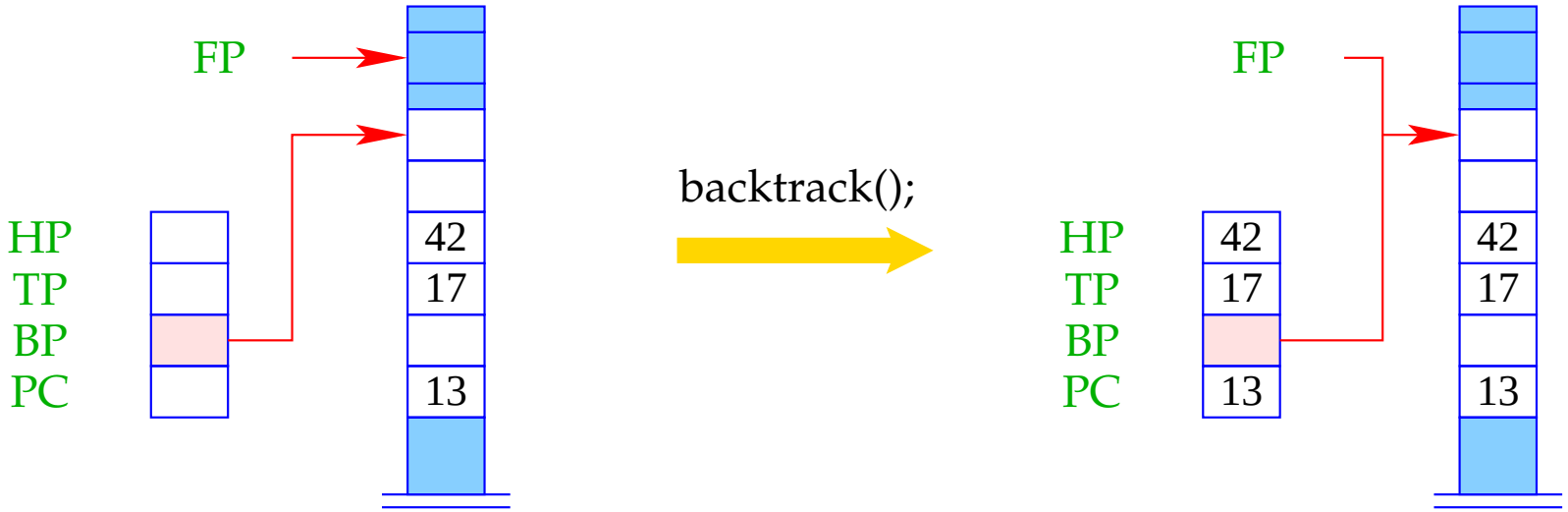
for the corresponding addresses.

Remark:

Occurrence on the left $\Longrightarrow$ saving the register

Occurrence on the right $\Longrightarrow$ restoring the register

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:

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

Note:

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

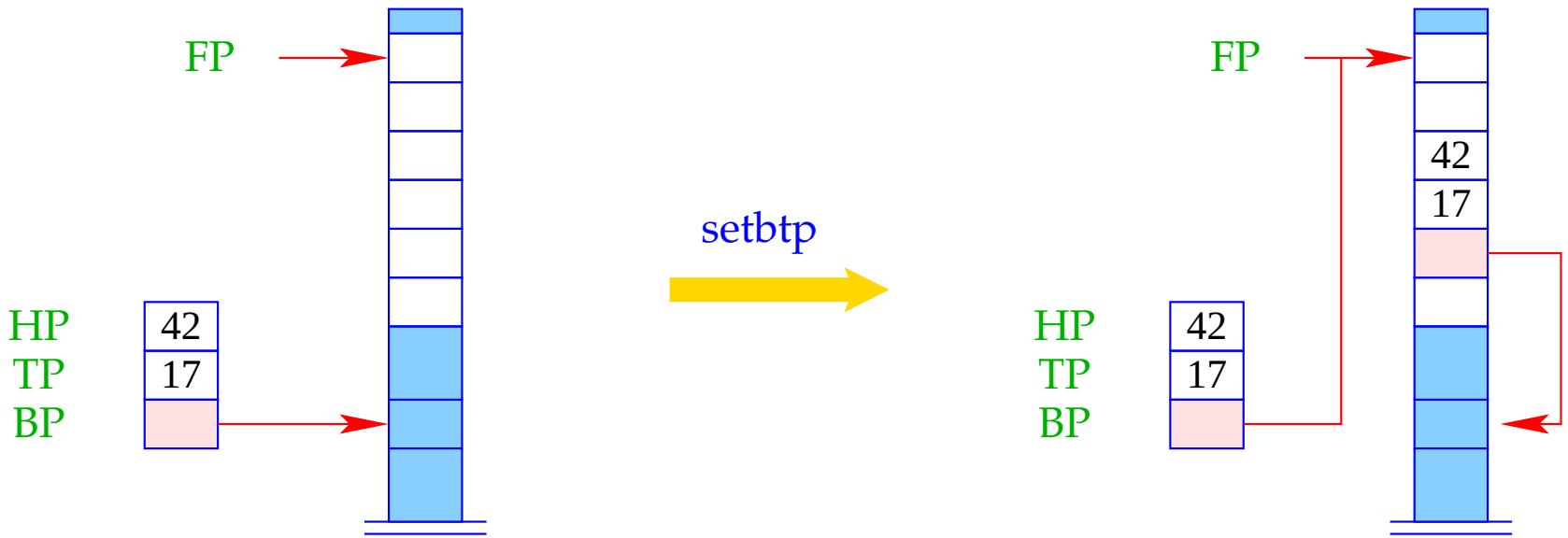
Example:

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

The translation of the predicate s yields:

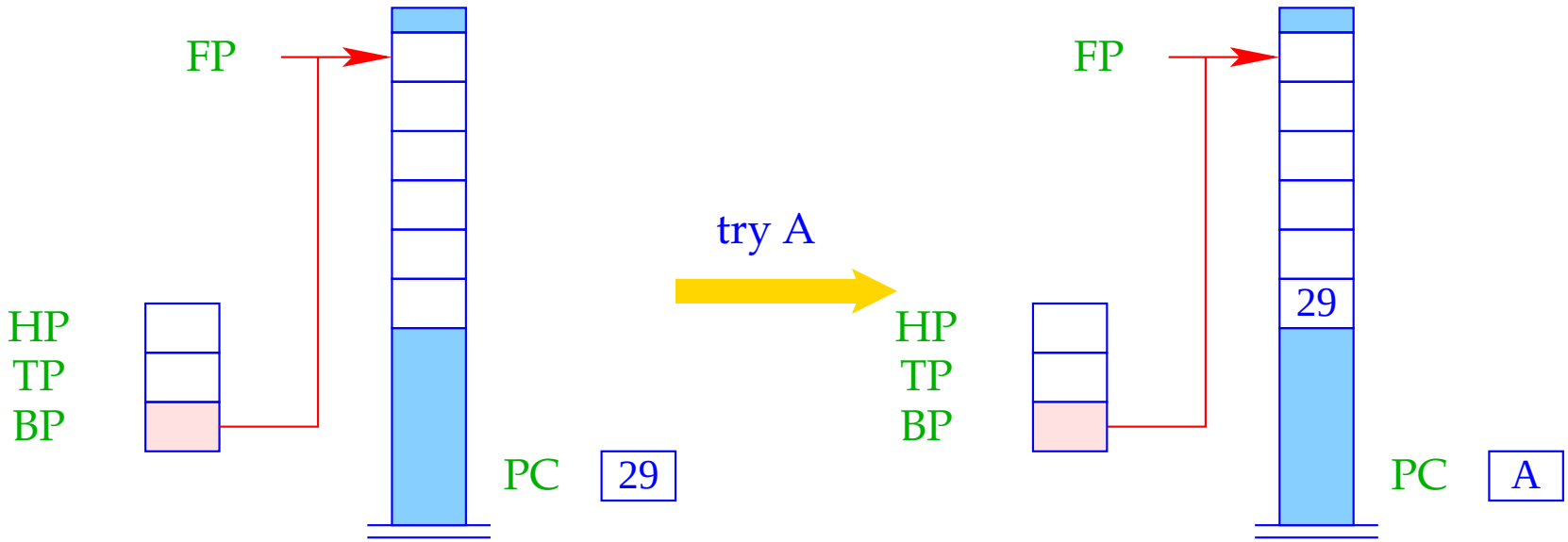
```
s/1:  setbtp      A:  pushenv 1     B:  pushenv 1
      try A           mark C            putref 1
      delbtp          putref 1          uatom a
      jump B          call t/1          popenv
                  C:  popenv
```

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



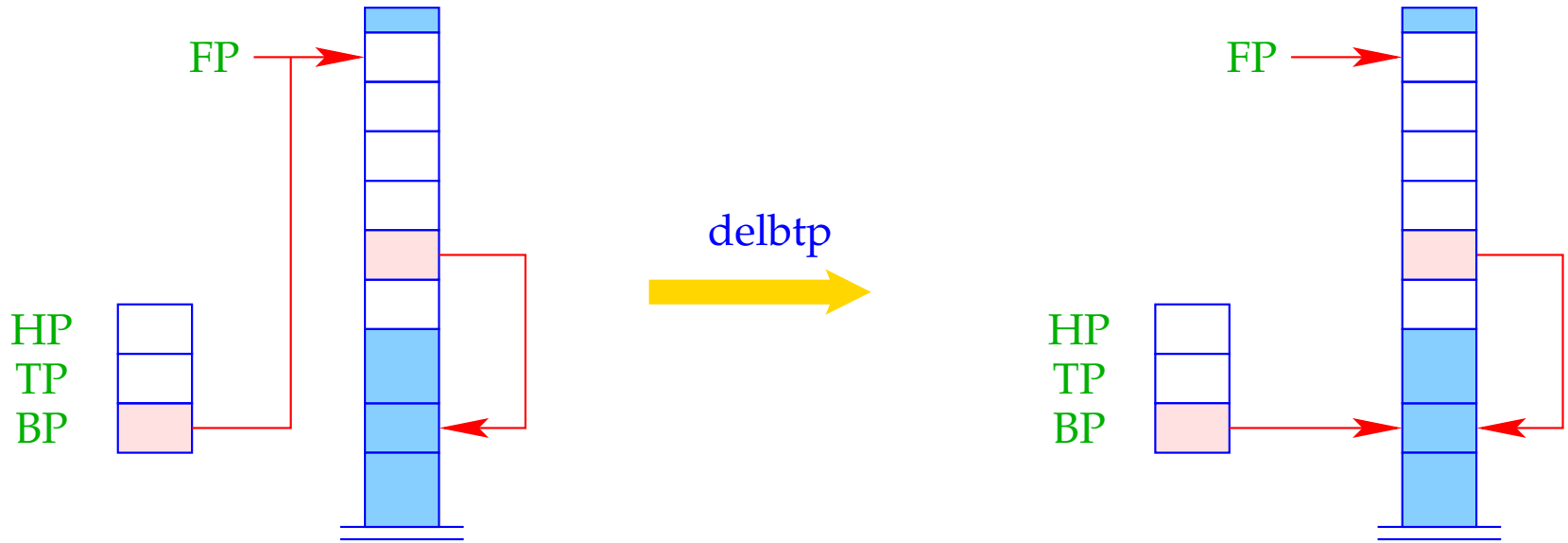

```
HPold = HP;
TPold = TP;
BPold = BP;
BP = FP;
```

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:

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

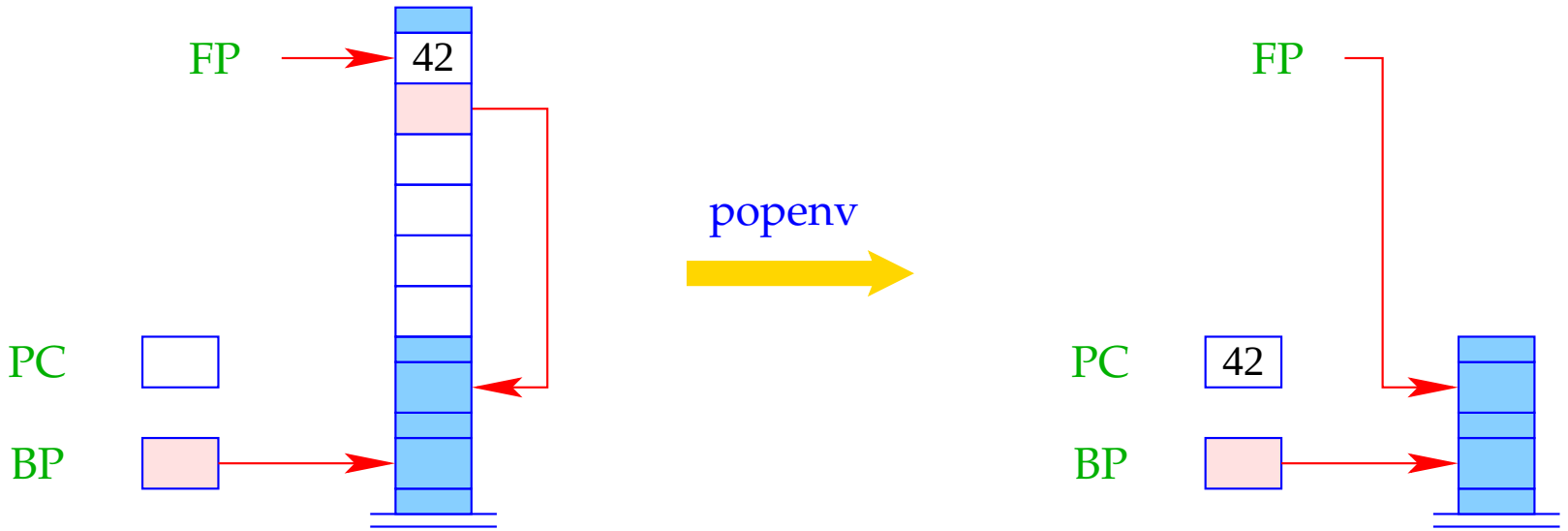
The present stack frame can be popped ...

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

$$\Longrightarrow \quad \text{the backtrack point is older} \quad \text{:-)}$$

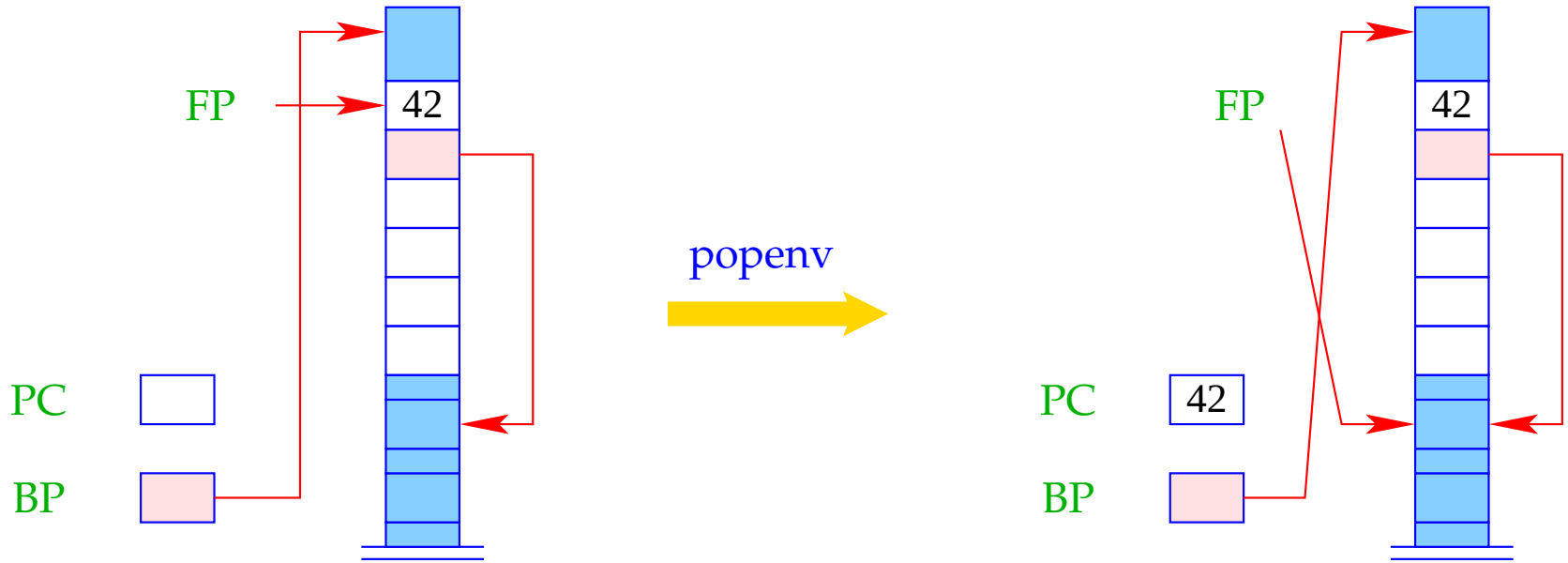
$$\Longrightarrow \quad \text{FP} > \text{BP}$$

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

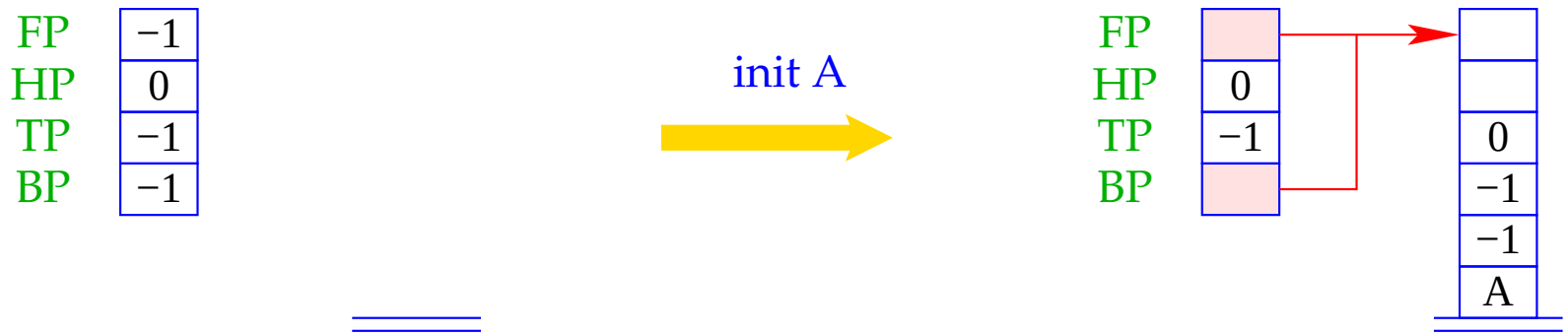
$$
\begin{array}{lll}
\text{code } p & = & \text{init A} \\
 & & \text{pushenv d} \\
 & & \text{code}_G\ g\ \rho \\
 & & \text{halt d} \\
 & \text{A:} & \text{no} \\
 & & \text{code}_P\ rr_1 \\
 & & \ldots \\
 & & \text{code}_P\ rr_h
\end{array}
$$

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          makr 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:

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

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: $p(X_1, \ldots, X_k) \leftarrow g_1, \ldots, g_n$

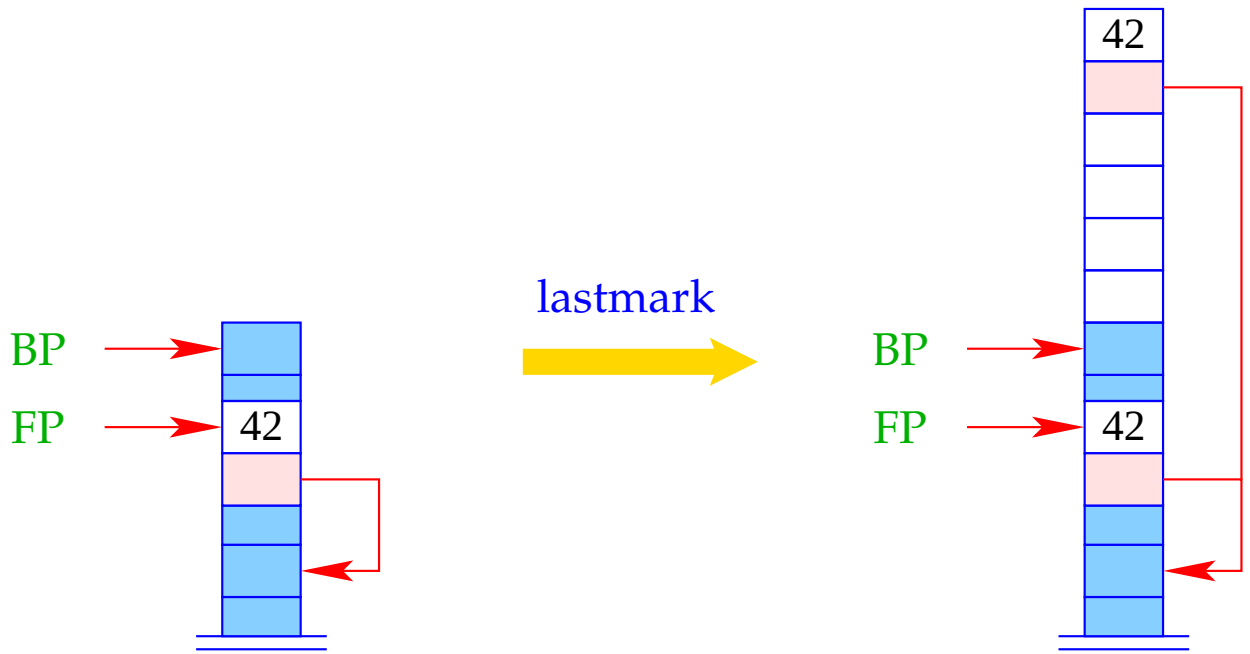
with m locals where $g_n \equiv q(t_1, \ldots, t_h)$. The interplay between code_C and code_G:

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

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 :-)

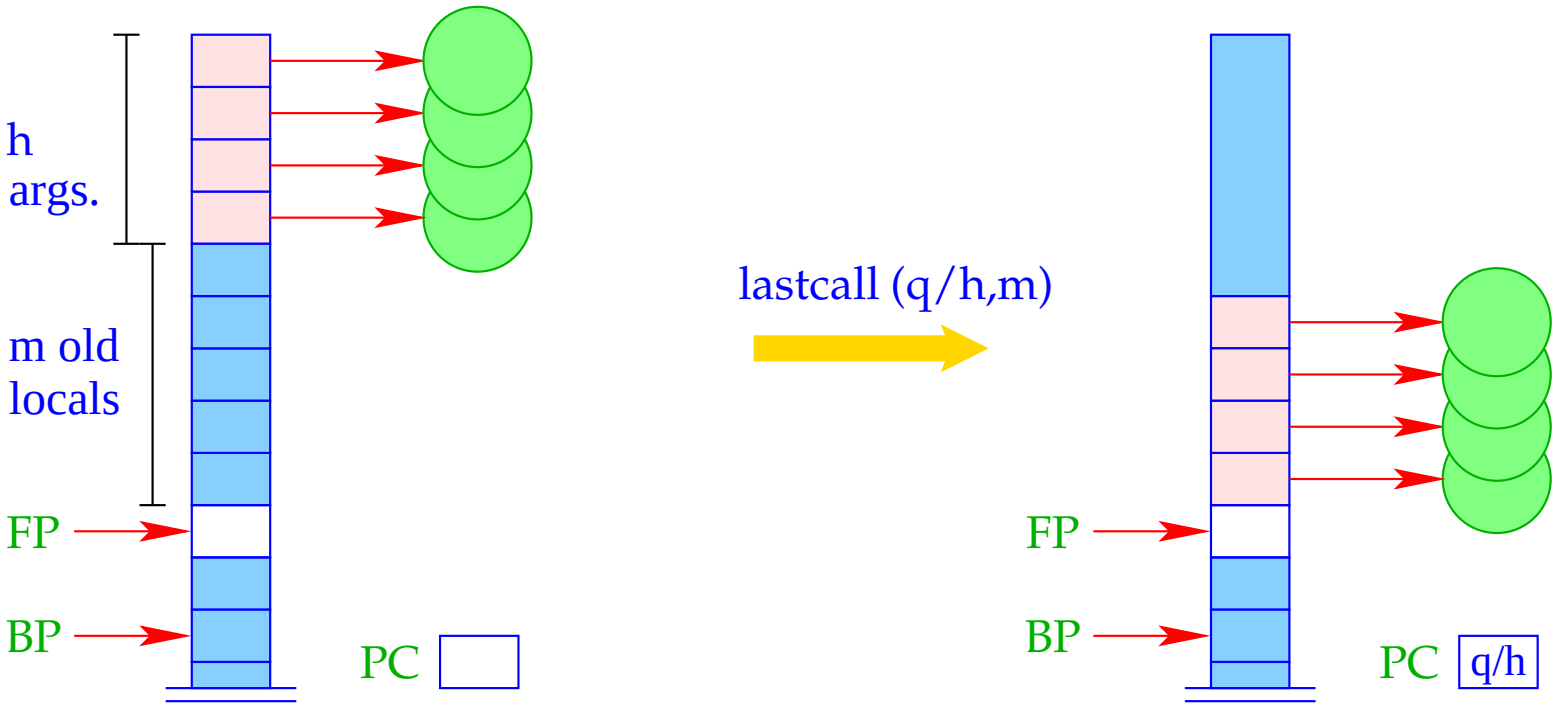
If FP $\leq$ BP, then lastcall q/h m behaves like a normal call q/h.

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
    bind

E:  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 :-}

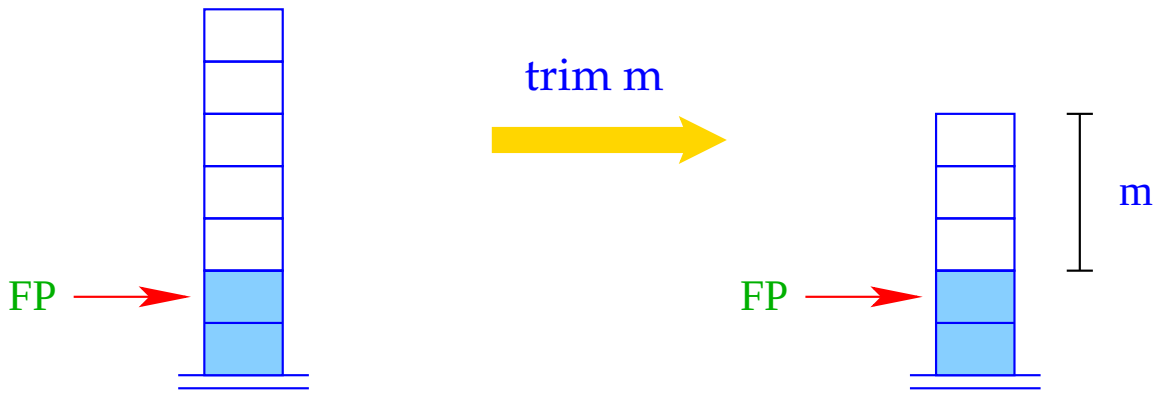
Example:

Consider the clause:

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

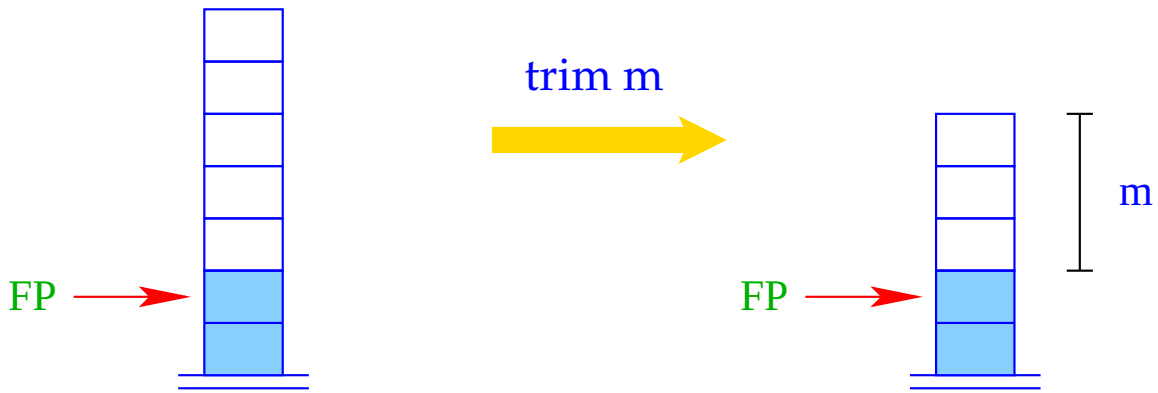
After the query $\mathsf{p}_2(\bar{X}_1, X_2)$, variable X_1 is dead.
After the query $\mathsf{p}_3(\bar{X}_2, X_3)$, variable X_2 is dead :-)

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			

36 Clause Indexing

Observation:

Often, predicates are implemented by case distinction on the first argument.

$\Longrightarrow$ Inspecting the first argument, many alternatives can be excluded :-)

$\Longrightarrow$ Failure is earlier detected :-)

$\Longrightarrow$ Backtrack points are earlier removed. :-))

$\Longrightarrow$ Stack frames are earlier popped :-)))

Example: The app-predicate:

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

- If the root constructor is $[\,]$, only the first clause is applicable.
- If the root constructor is $[|]$, only the second clause is applicable.
- Every other root constructor should fail !!
- Only if the first argument equals an unbound variable, both alternatives must be tried ;-)

Idea:

- Introduce separate try chains for every possible constructor.
- Inspect the root node of the first argument.
- Depending on the result, perform an indexed jump to the appropriate try chain.

Assume that the predicate p/k is defined by the sequence rr of clauses $r_1 \ldots r_m$.

Let tchains rr denote the sequence of try chains as built up for the root constructors occurring in unifications $X_1 = t$.

Example:

Consider again the app-predicate, and assume that the code for the two clauses start at addresses A_1 and A_2, respectively.

Then we obtain the following four try chains:

```
VAR:   setbtp      // variables     NIL:    jump A1     // atom [ ]
       try A1
       delbtp                     CONS:   jump A2     // constructor [|]
       jump A2
                                  ELSE:   fail        // default
```

Example:

Consider again the app-predicate, and assume that the code for the two clauses start at addresses A_1 and A_2, respectively.

Then we obtain the following four try chains:

```
VAR:   setbtp      // variables     NIL:    jump A1     // atom [ ]
       try A1
       delbtp                     CONS:   jump A2     // constructor [|]
       jump A2
                                  ELSE:   fail        // default
```

The new instruction fail takes care of any constructor besides [ ] and [|] ...

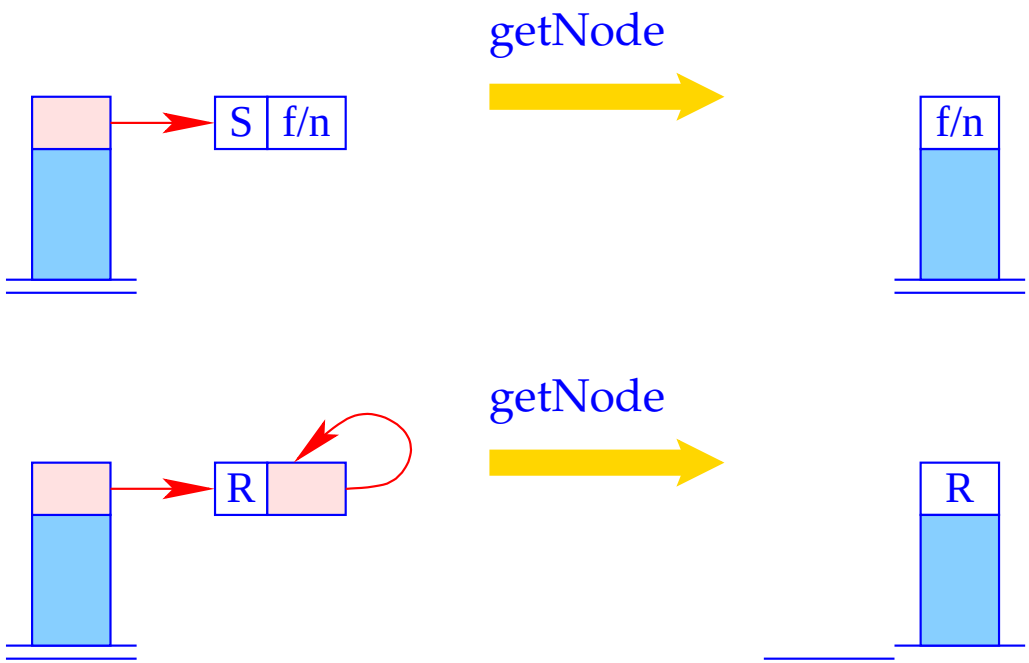
```
fail  =  backtrack()
```

It directly triggers backtracking :-)

Then we generate for a predicate p/k:

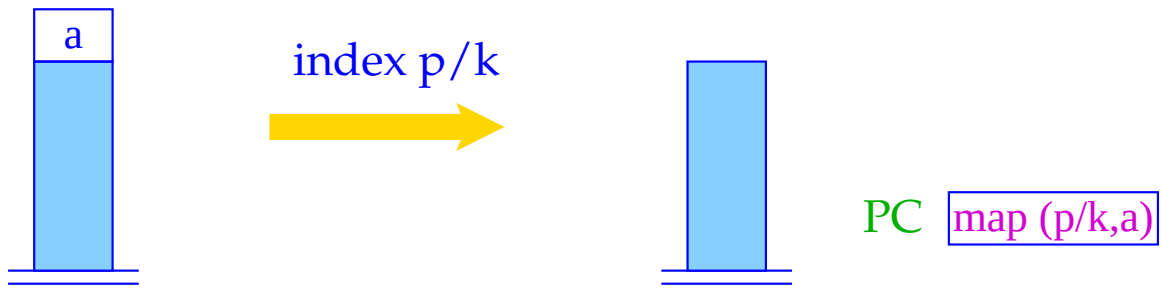
$$
\begin{array}{lll}
\text{code}_P\ rr = & \text{putref 1} & \\
 & \text{getNode} & \text{// extracts the root label} \\
 & \text{index p/k} & \text{// jumps to the try block} \\
 & \text{tchains } rr & \\
A_1: & \text{code}_C\ r_1 & \\
 & \ldots & \\
A_m: & \text{code}_C\ r_m &
\end{array}
$$

The instruction getNode returns "R" if the pointer on top of the stack points to an unbound variable. Otherwise, it returns the content of the heap object:



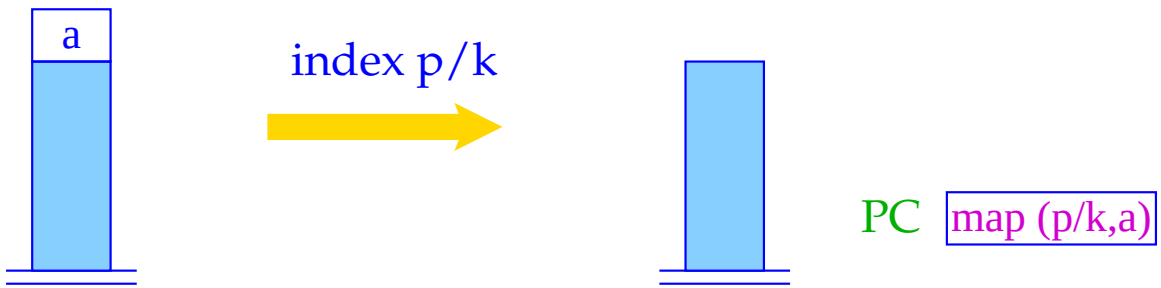

```
switch (H[S[SP]]) {
case (S, f/n):    S[SP] = f/n; break;
case (A,a):       S[SP] = a; break;
case (R,_) :      S[SP] = R;
}
```

The instruction index p/k performs an indexed jump to the appropriate try chain:




```
PC = map (p/k,S[SP]);
SP– –;
```

The instruction index p/k performs an indexed jump to the appropriate try chain:




```
PC = map (p/k,S[SP]);
SP– –;
```

The function map() returns, for a given predicate and node content, the start address of the appropriate try chain :-)

It typically is defined through some hash table :-))

37 Extension: The Cut Operator

Realistic Prolog additionally provides an operator "!" (cut) which explicitly allows to prune the search space of backtracking.

Example:

$$\begin{aligned} \mathsf{branch}(X, Y) &\leftarrow \mathsf{p}(X), !, \mathsf{q}_1(X, Y) \\ \mathsf{branch}(X, Y) &\leftarrow \mathsf{q}_2(X, Y) \end{aligned}$$

Once the queries before the cut have succeeded, the choice is committed:

Backtracking will return only to backtrack points preceding the call to the left-hand side ...

The Basic Idea:

- We restore the oldBP from our current stack frame;
- We pop all stack frames on top of the local variables.

Accordingly, we translate the cut into the sequence:

```
prune
pushenv m
```

where `m` is the number of (still used) local variables of the clause.

Example:

Consider our example:

$$\begin{array}{lcl} \mathsf{branch}(X,Y) & \leftarrow & \mathsf{p}(X), !, \mathsf{q}_1(X,Y) \\ \mathsf{branch}(X,Y) & \leftarrow & \mathsf{q}_2(X,Y) \end{array}$$

We obtain:

```
setbtp      A:  pushenv 2     C:  prune          lastmark        B:  pushenv 2
try A           mark C            pushenv 2      putref 1            putref 2
delbtp          putref 1                         putref 2            putref 2
jump B          call p/1                         lastcall q1/2 2     move 2 2
                                                                     jump q2/2
```

Example:

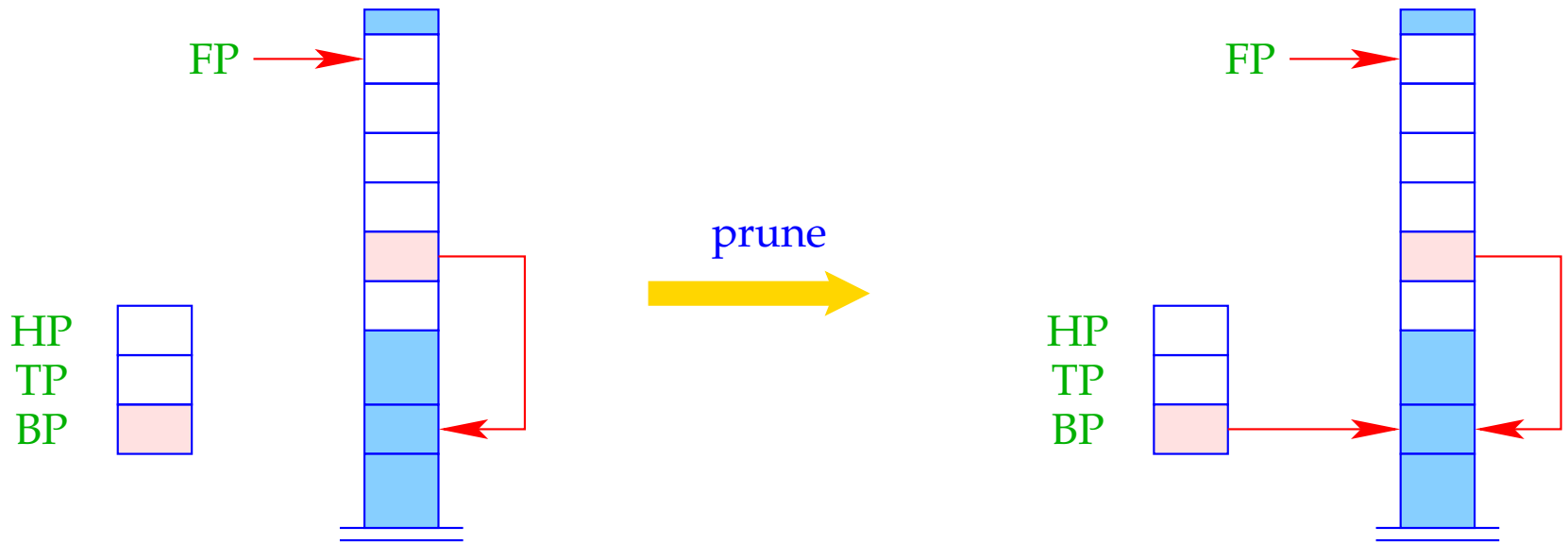
Consider our example:

$$\begin{aligned} \mathsf{branch}(X,Y) &\leftarrow \mathsf{p}(X), !, \mathsf{q}_1(X,Y) \\ \mathsf{branch}(X,Y) &\leftarrow \mathsf{q}_2(X,Y) \end{aligned}$$

In fact, an optimized translation even yields here:

```
setbtp      A:  pushenv 2     C:  prune       putref 1      B:  pushenv 2
try A           mark C            pushenv 2   putref 2          putref 1
delbtp          putref 1                      move 2 2          putref 2
jump B          call p/1                      jump q1/2         move 2 2
                                                                jump q2/2
```

The new instruction prune simply restores the backtrack pointer:



BP = BPold;

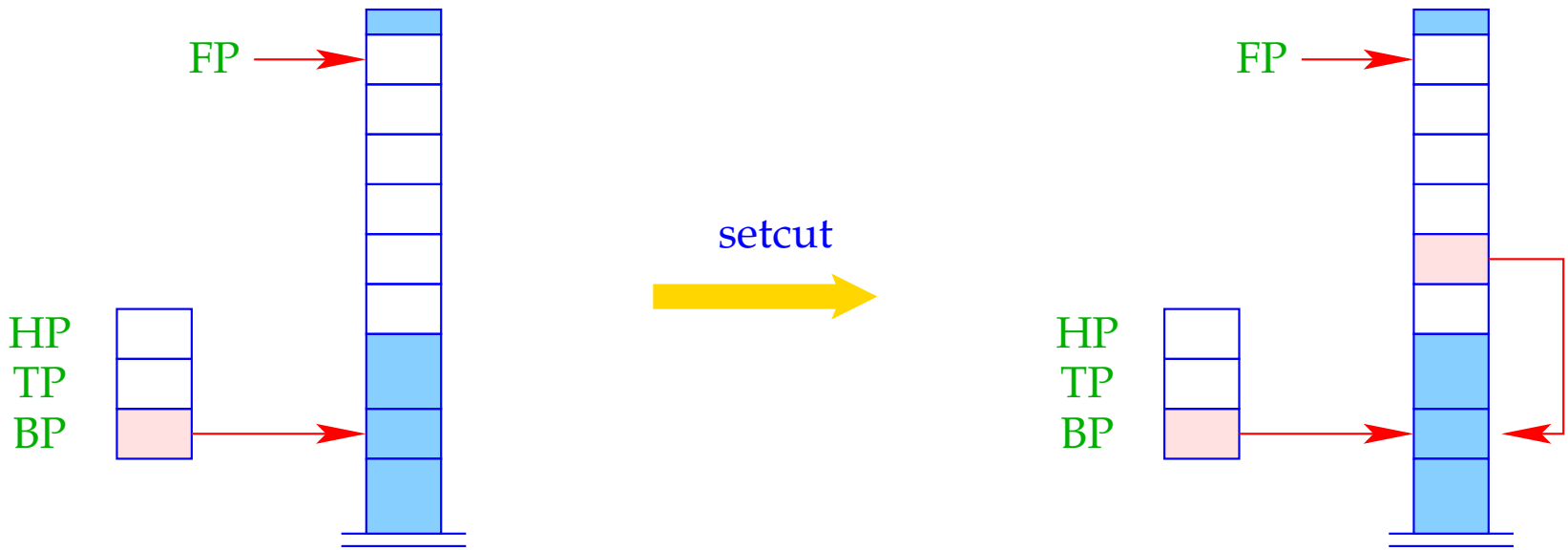
Problem:

If a clause is single, then (at least so far ;-) we have not stored the old BP inside the stack frame :-(

$\Longrightarrow$

For the cut to work also with single-clause predicates or try chains of length 1, we insert an extra instruction setcut before the clausal code (or the jump):

The instruction setcut just stores the current value of BP:



BPold = BP;

The Final Example: Negation by Failure

The predicate notP should succeed whenever p fails (and vice versa :-)

$$\begin{aligned} \mathsf{notP}(X) &\leftarrow \mathsf{p}(X), !, \mathsf{fail} \\ \mathsf{notP}(X) &\leftarrow \end{aligned}$$

where the goal fail never succeeds. Then we obtain for notP :

```
setbtp      A:  pushenv 1     C:  prune       B:  pushenv 1
try A           mark C            pushenv 1       popenv
delbtp          putref 1          fail
jump B          call p/1          popenv
```

38 Garbage Collection

- Both during execution of a MaMa- as well as a WiM-programs, it may happen that some objects can no longer be reached through references.
- Obviously, they cannot affect the further program execution. Therefore, these objects are called garbage.
- Their storage space should be freed and reused for the creation of other objects.

Warning:

The WiM provides some kind of heap de-allocation. This, however, only frees the storage of failed alternatives !!!

Operation of a stop-and-copy-Collector:

- Division of the heap into two parts, the to-space and the from-space — which, after each collection flip their roles.
- Allocation with new in the current from-space.
- In case of memory exhaustion, call of the collector.

The Phases of the Collection:

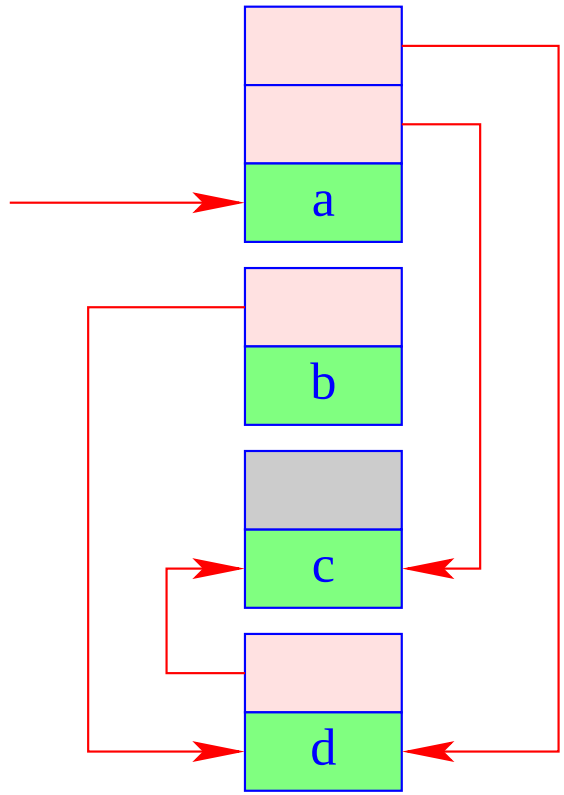
1. Marking of all reachable objects in the from-space.
2. Copying of all marked objects into the to-space.
3. Correction of references.
4. Exchange of from-space and to-space.

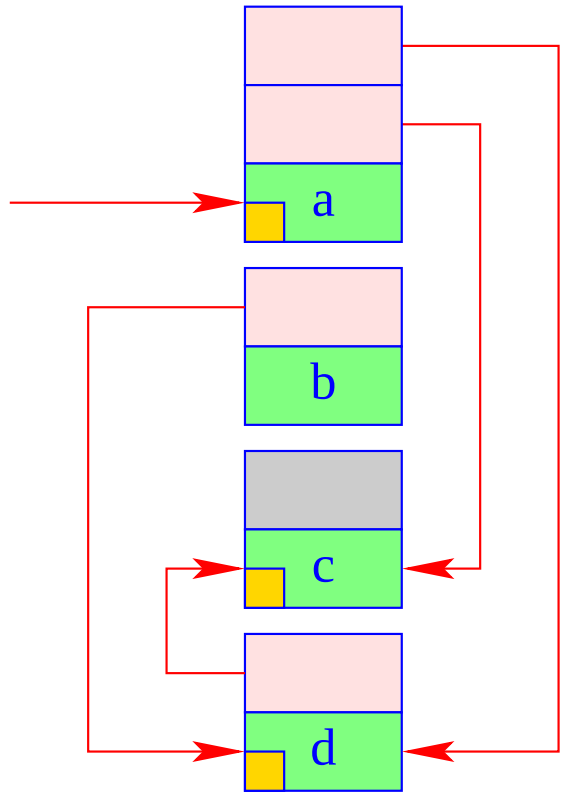
(1) Mark: Detection of live objects:

- all references in the stack point to live objects;
- every reference of a live object points to a live object.

$\Longrightarrow$

Graph Reachability


a
b
c
d

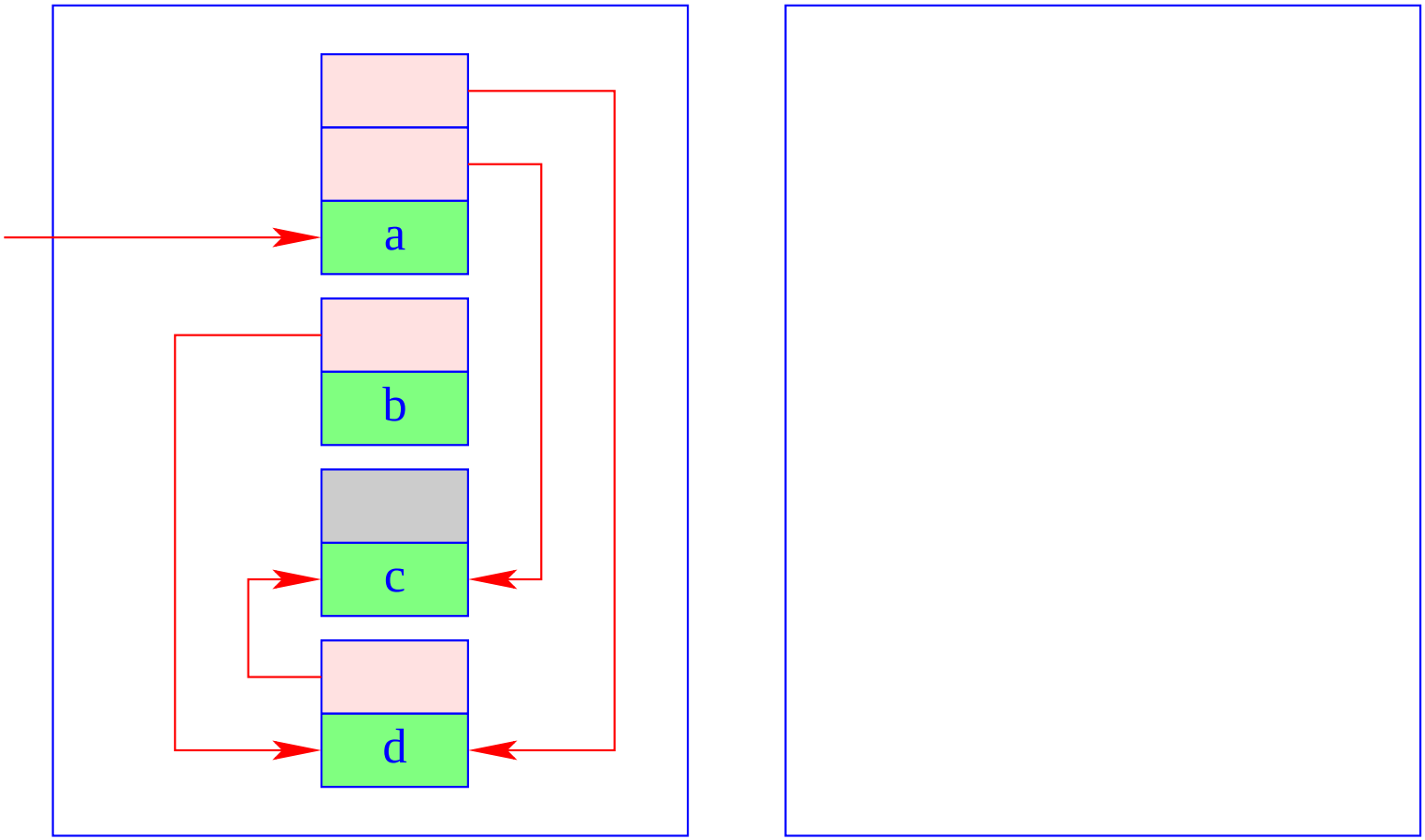

a
b
c
d

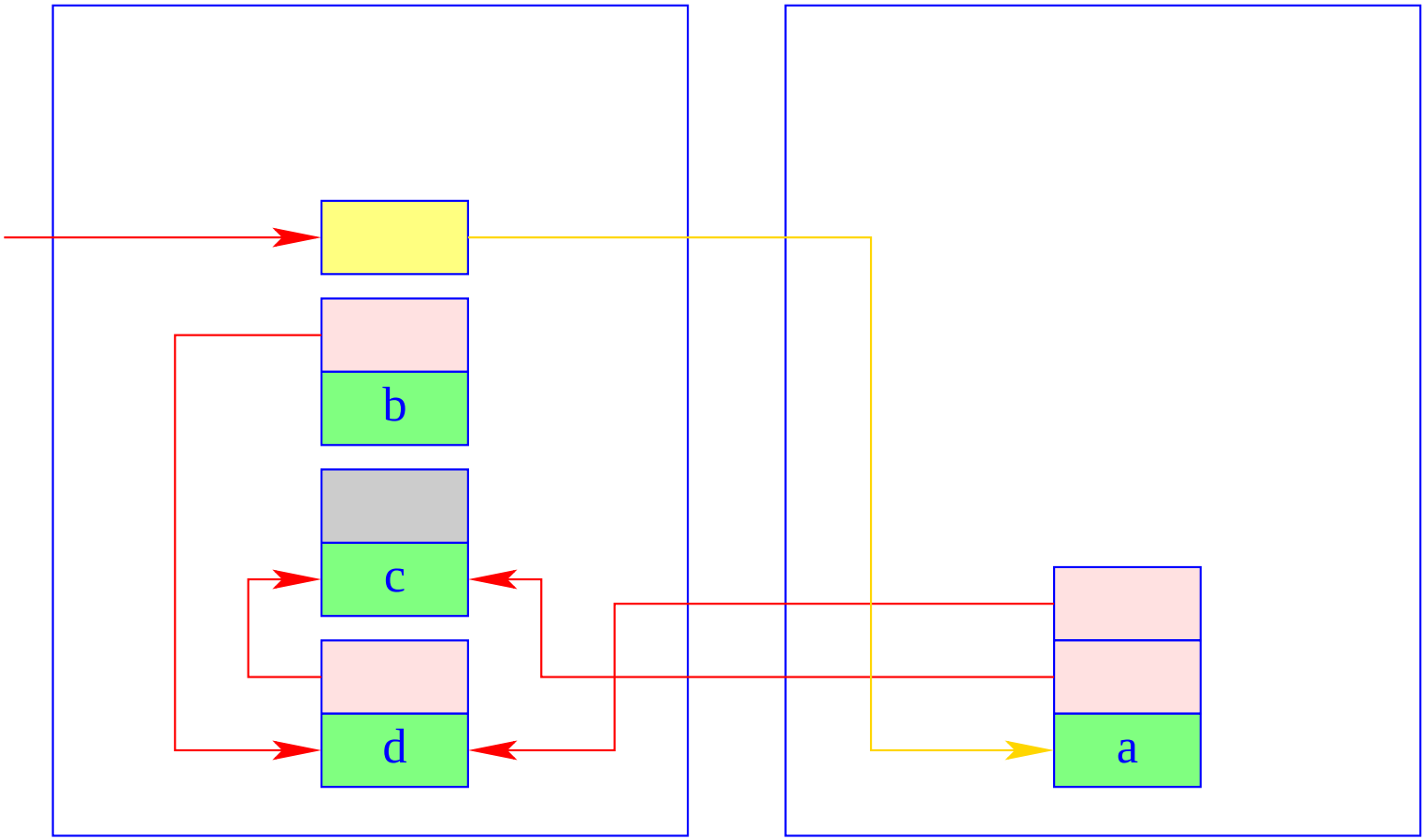
(2) **Copy:** Copying of all live objects from the current from-space into the current to-space. This means for every detected object:

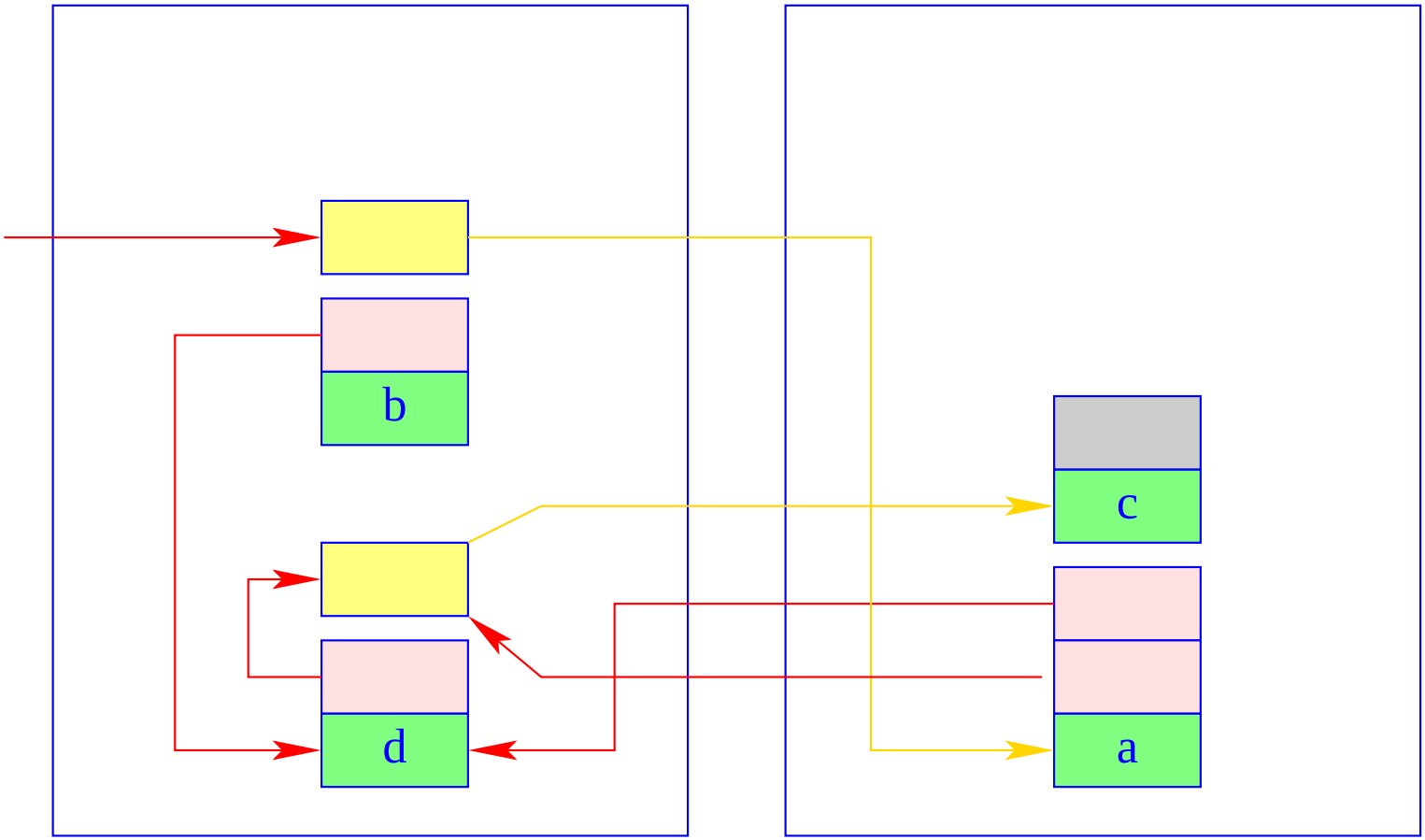
- Copying the object;
- Storing a forward reference to the new place at the old place :-)

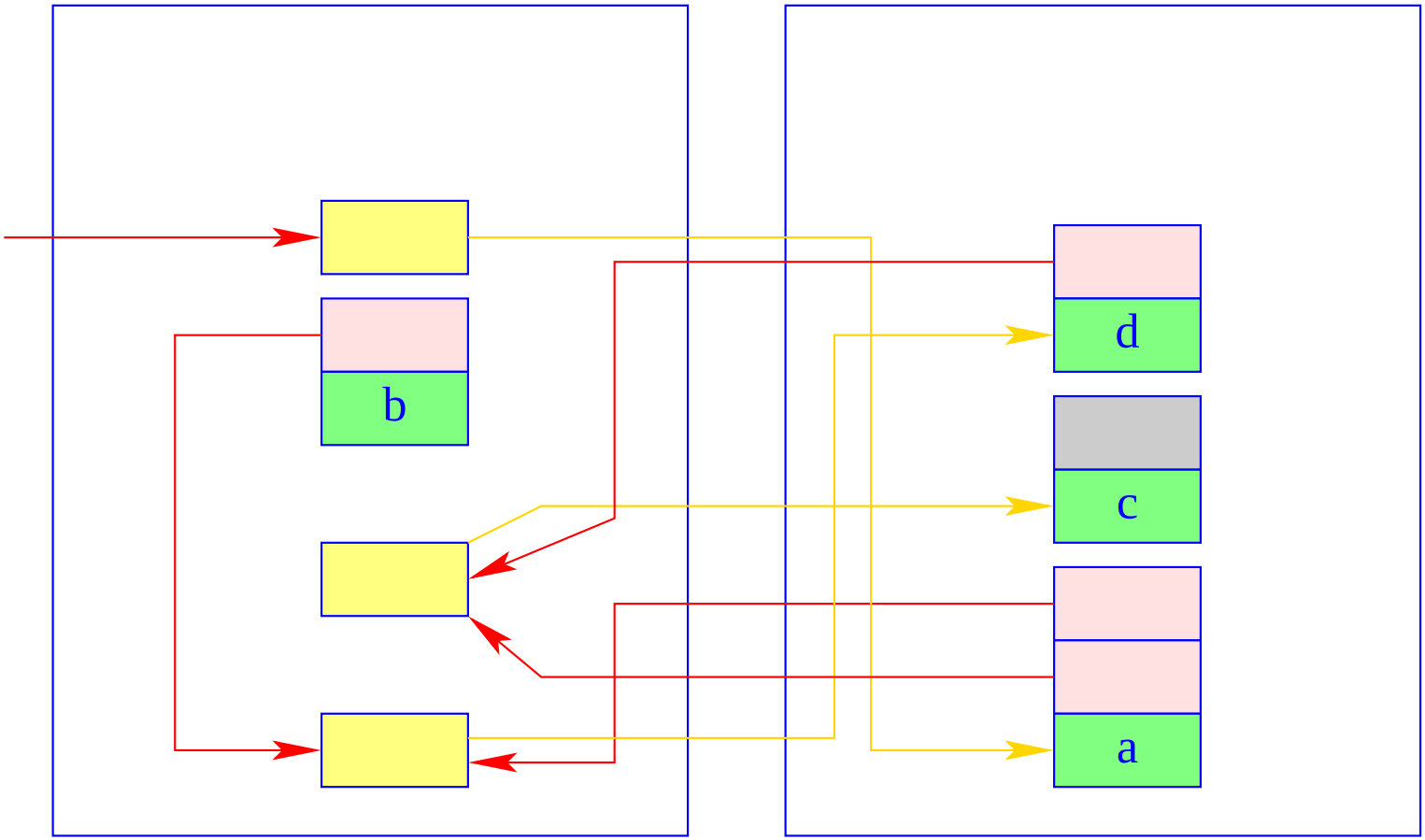
$$\Longrightarrow$$

all references of the copied objects point to the forward references in the from-space.

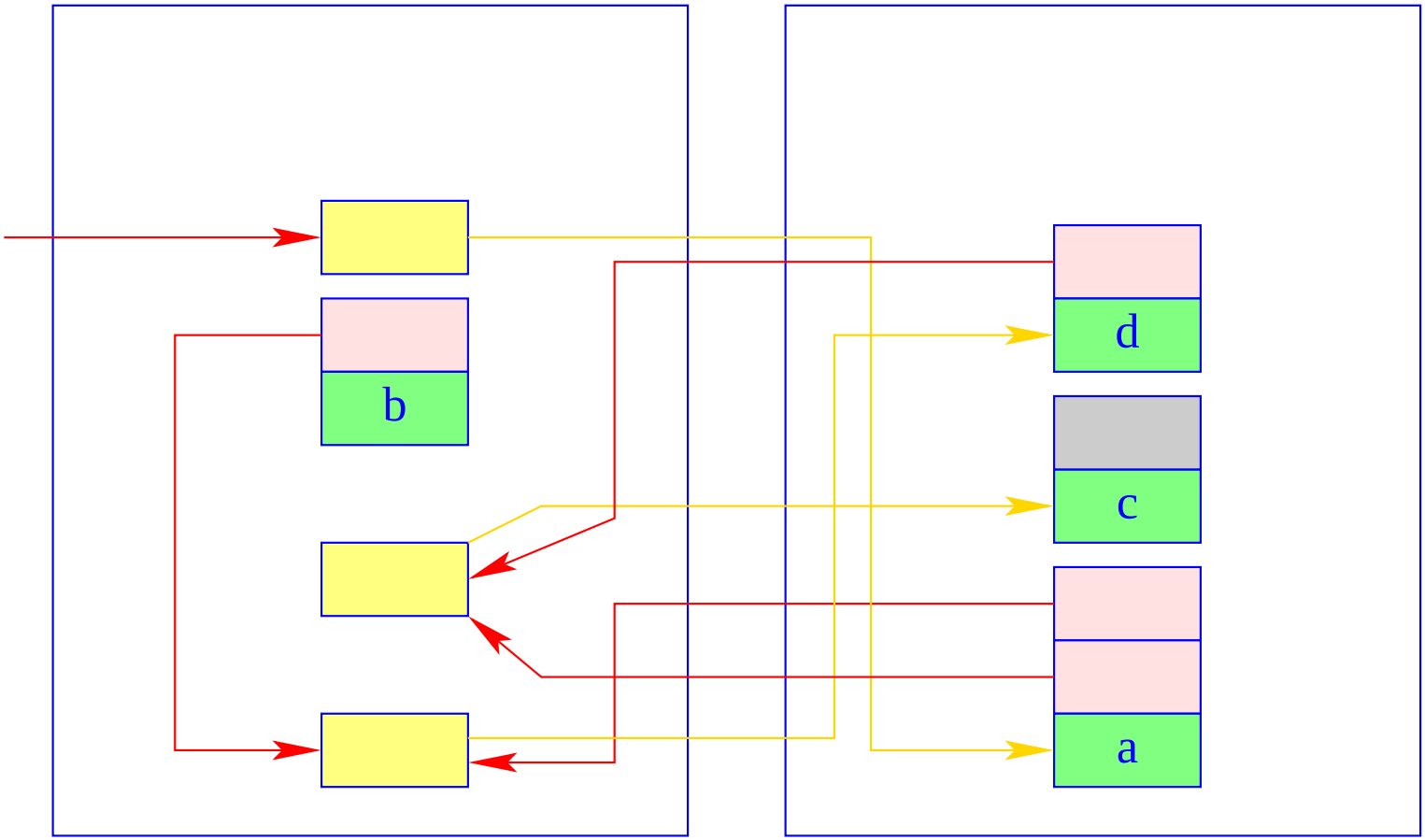

a
b
c
d

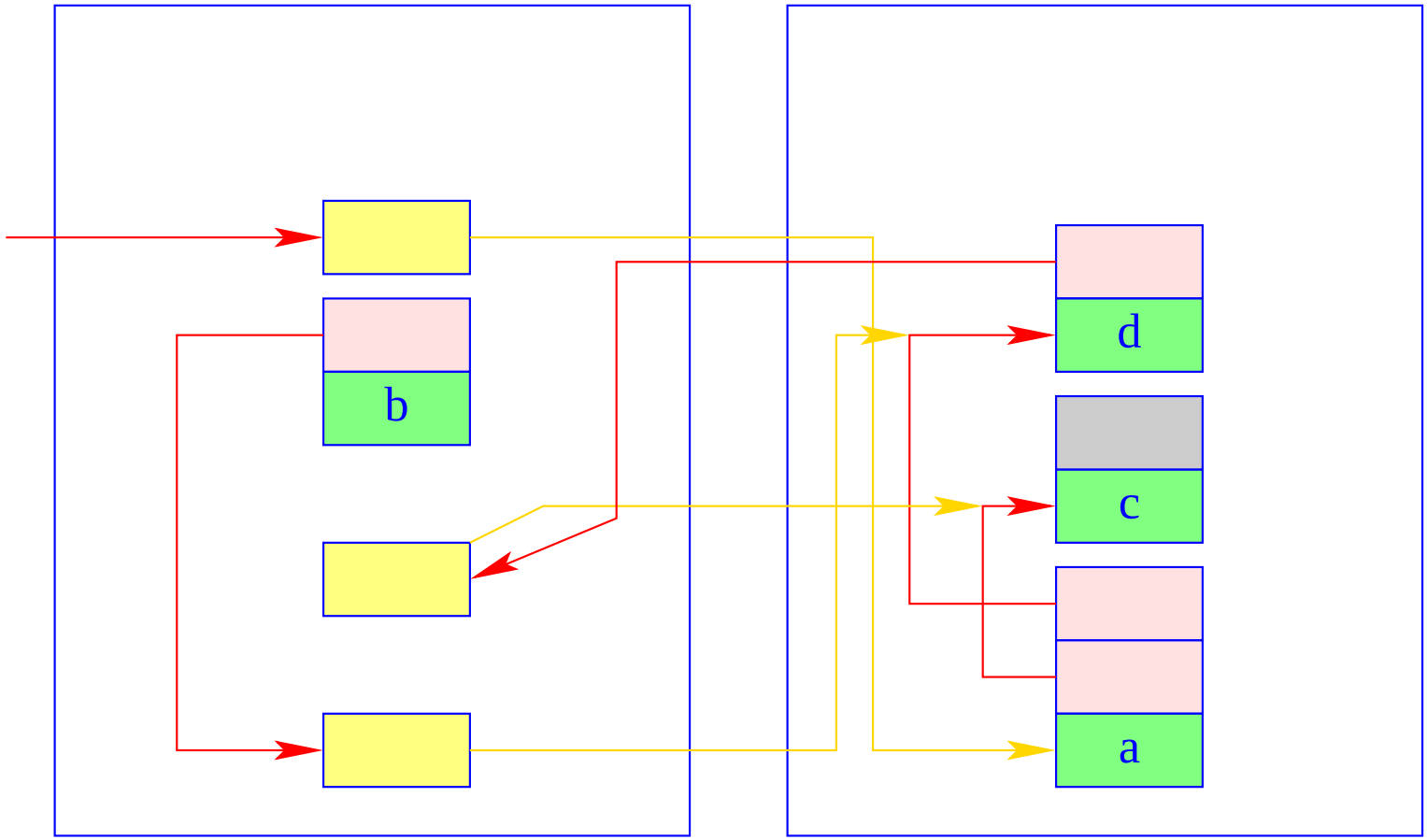

b
c
d
a

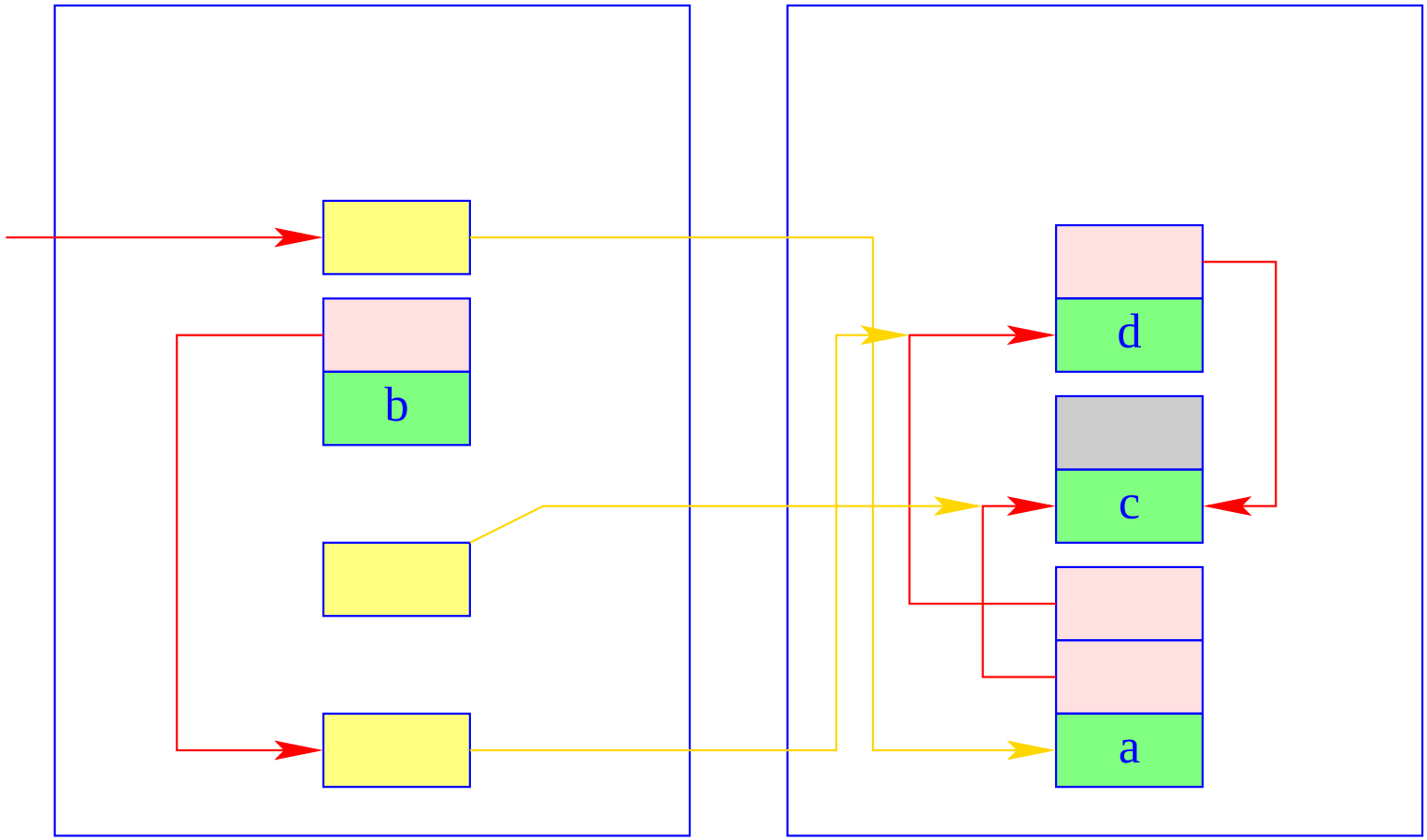

b
c
d
a

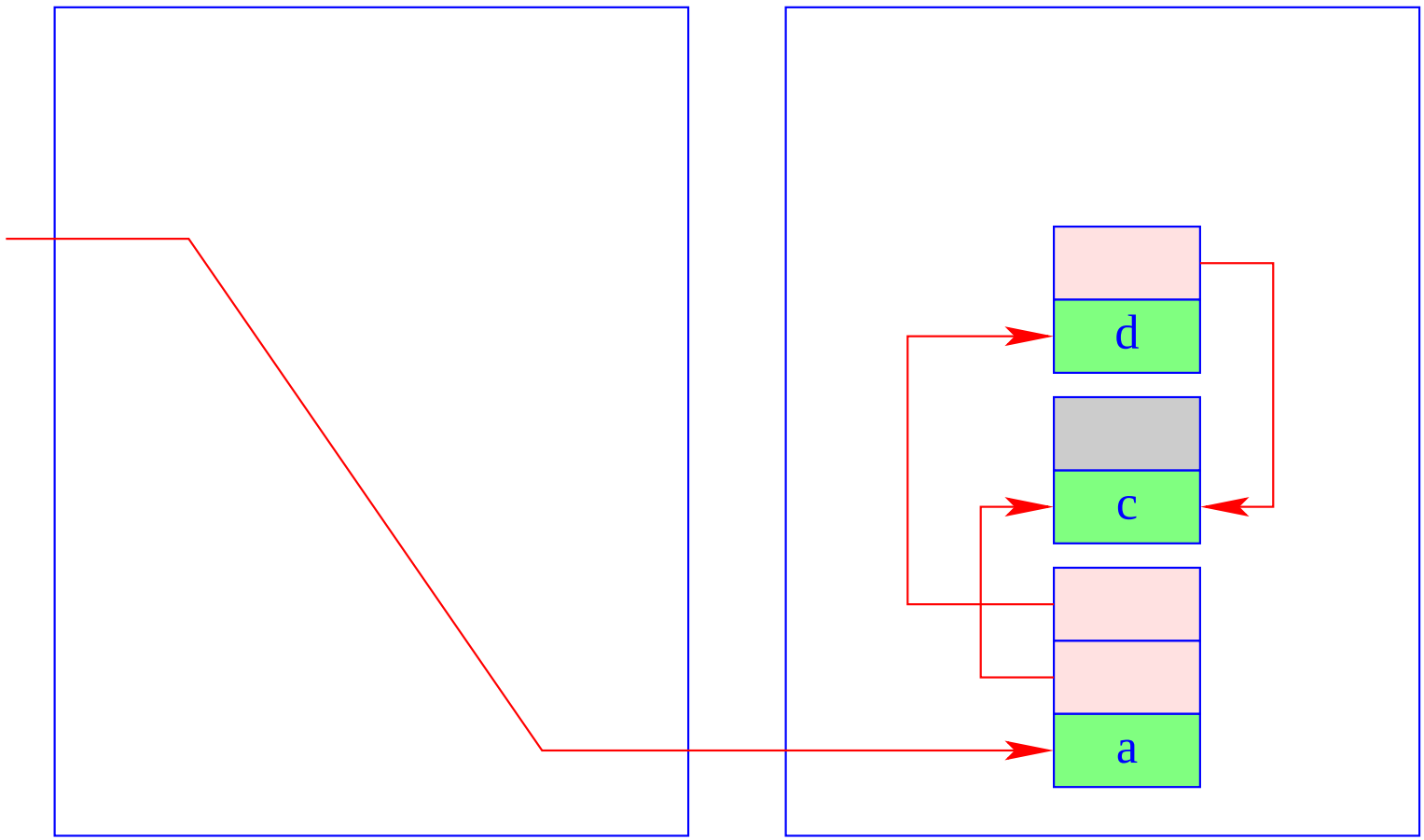

b
d
c
a

(3) Traversing of the to-space in order to correct the references.

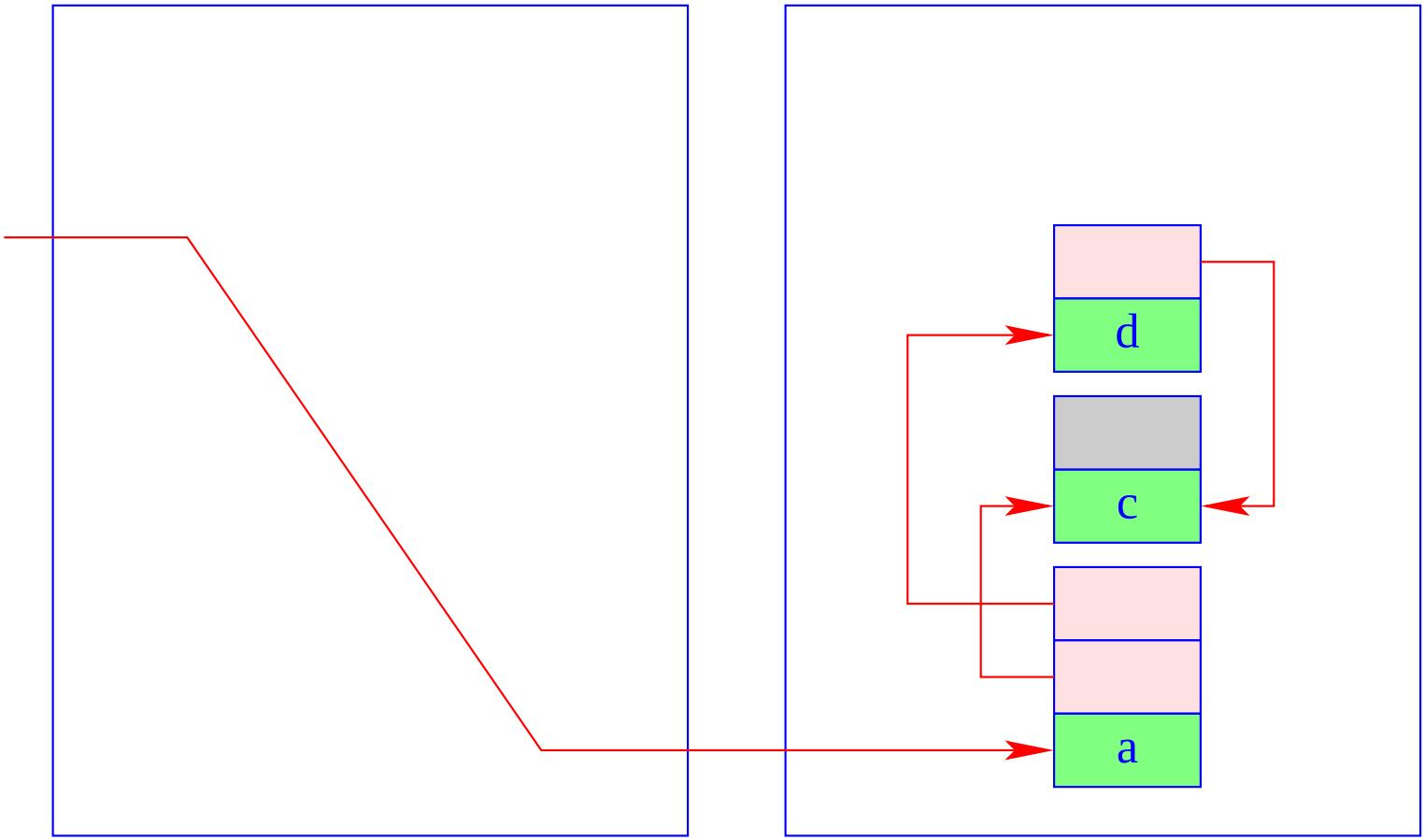


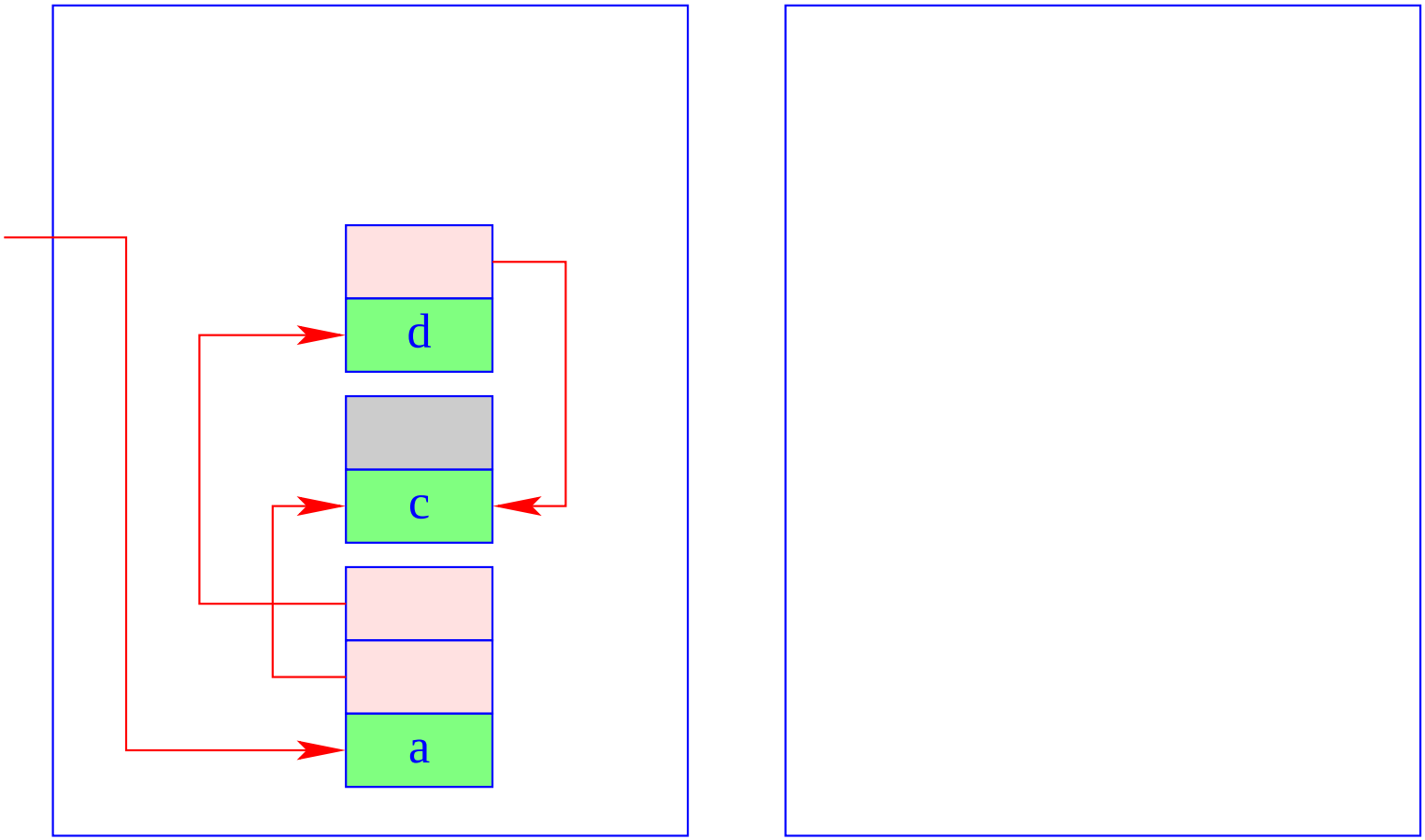

b
d
c
a


b
d
c
a


d
c
a

(4) Exchange of to-space and from-space.



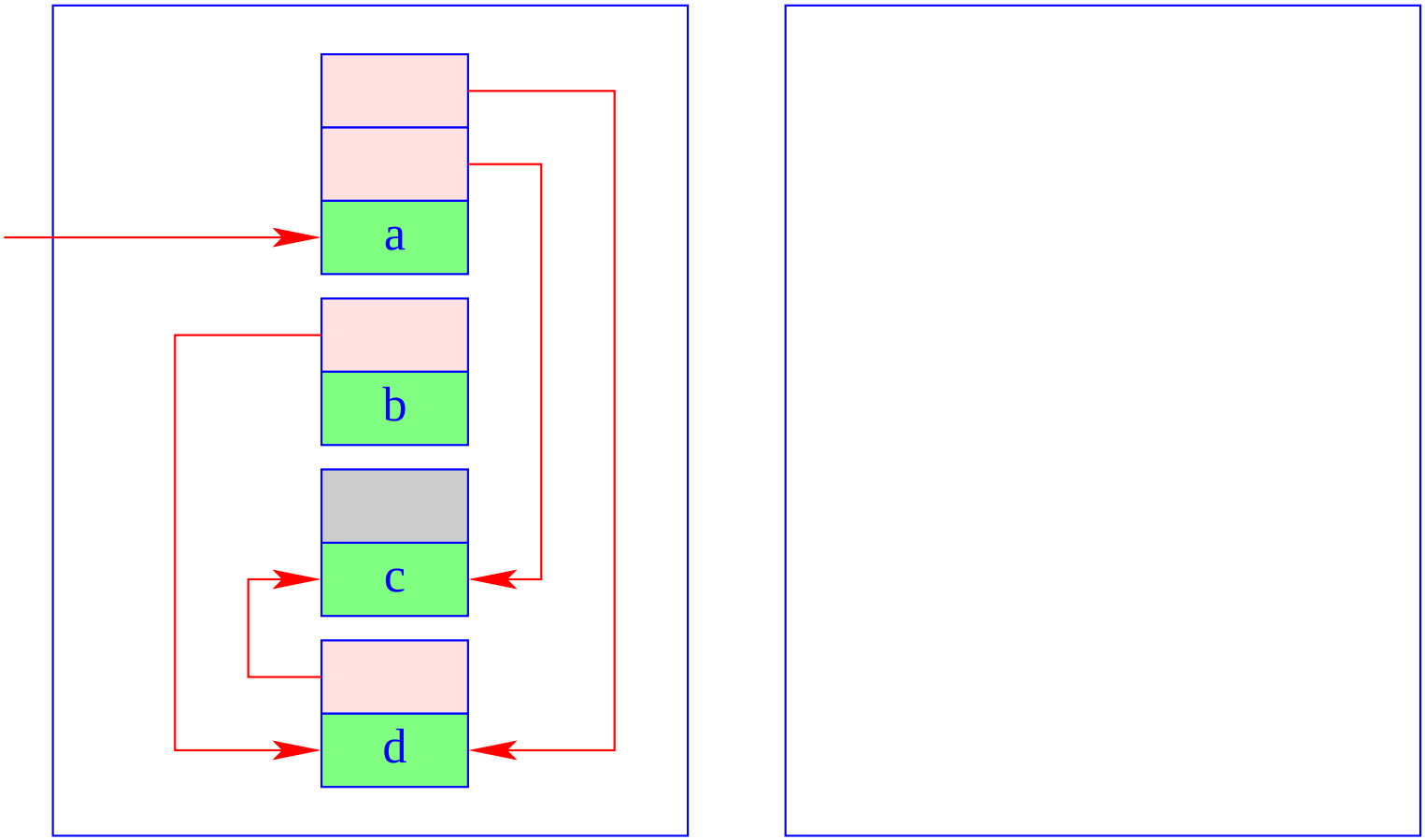

d
c
a

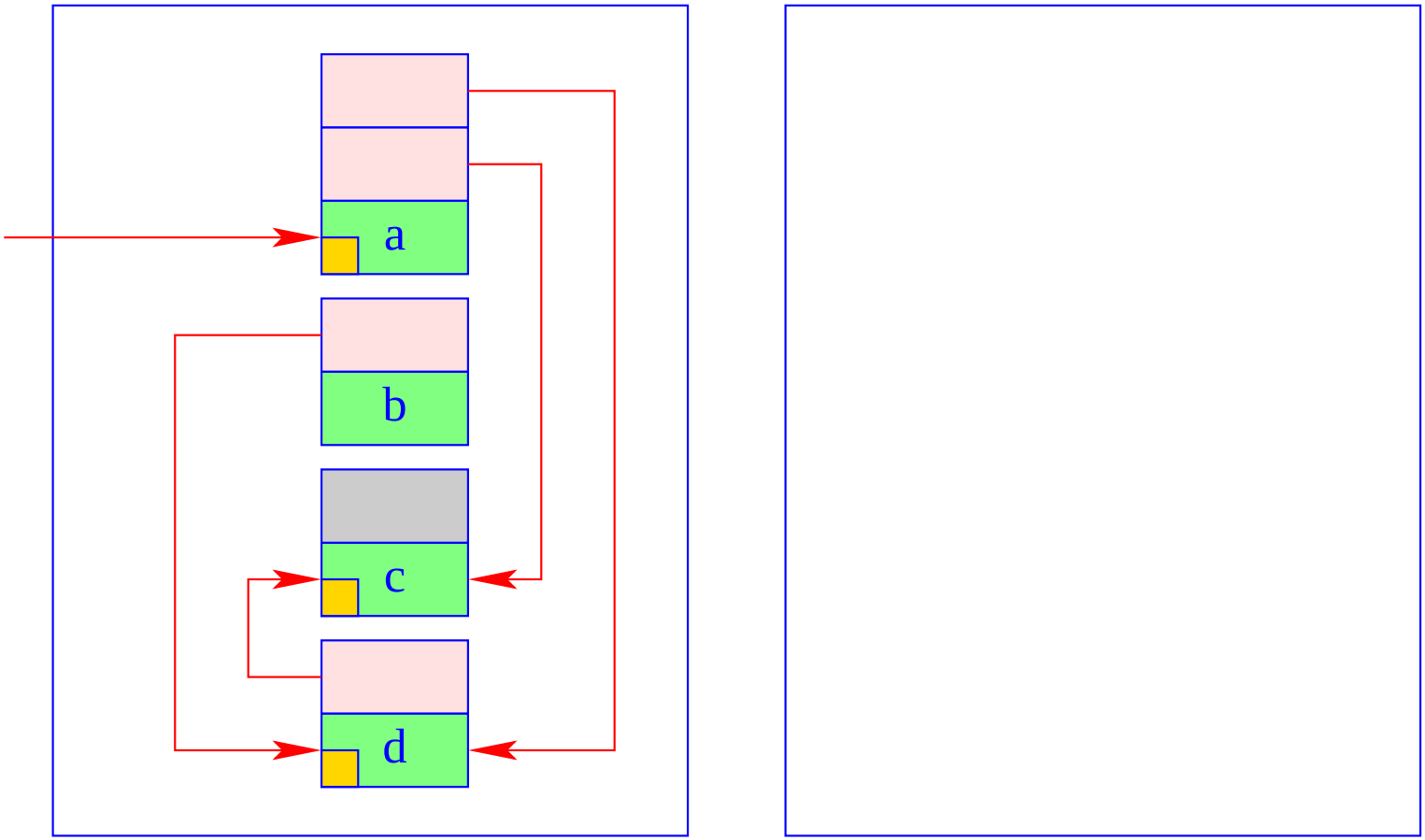
Warning:

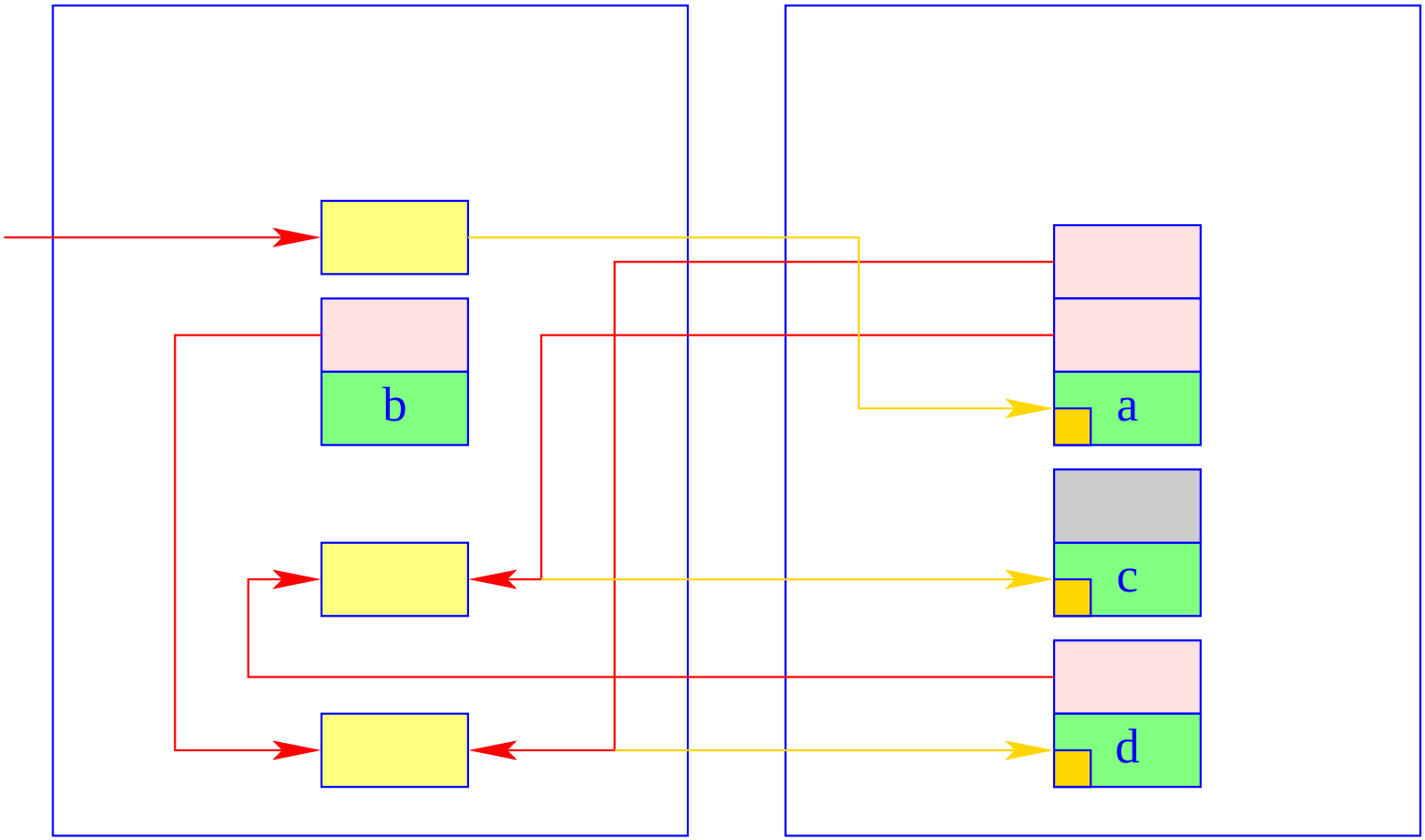
The garbage collection of the WiM must harmonize with backtracking.

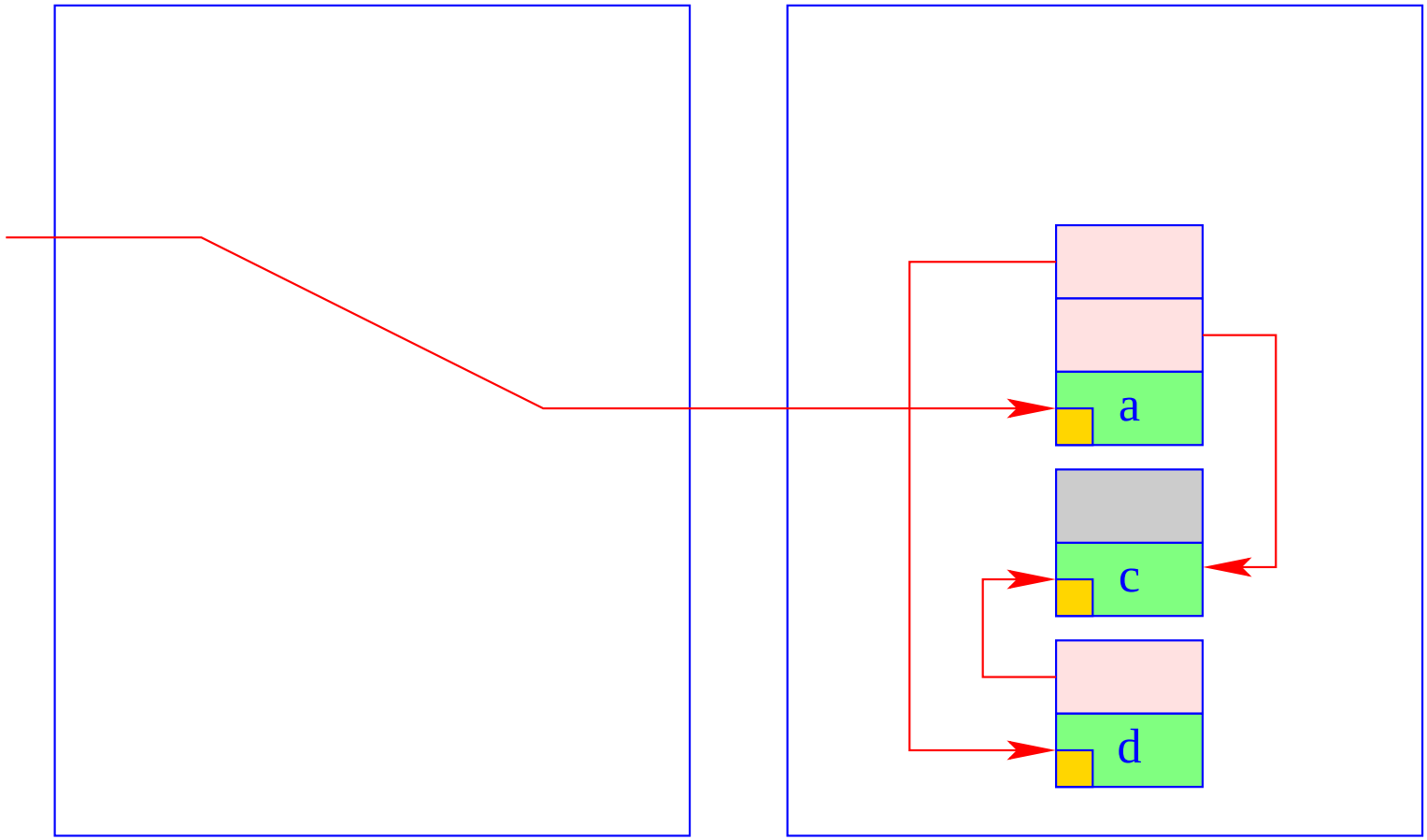
This means:

- The relative position of heap objects must not change during copying :-!!
- The heap references in the trail must be updated to the new positions.
- If heap objects are collected which have been created before the last backtrack point, then also the heap pointers in the stack must be updated.


a
b
c
d


a
b
c
d


b
a
c
d


a
c
d

Threads

39 The Language ThreadedC

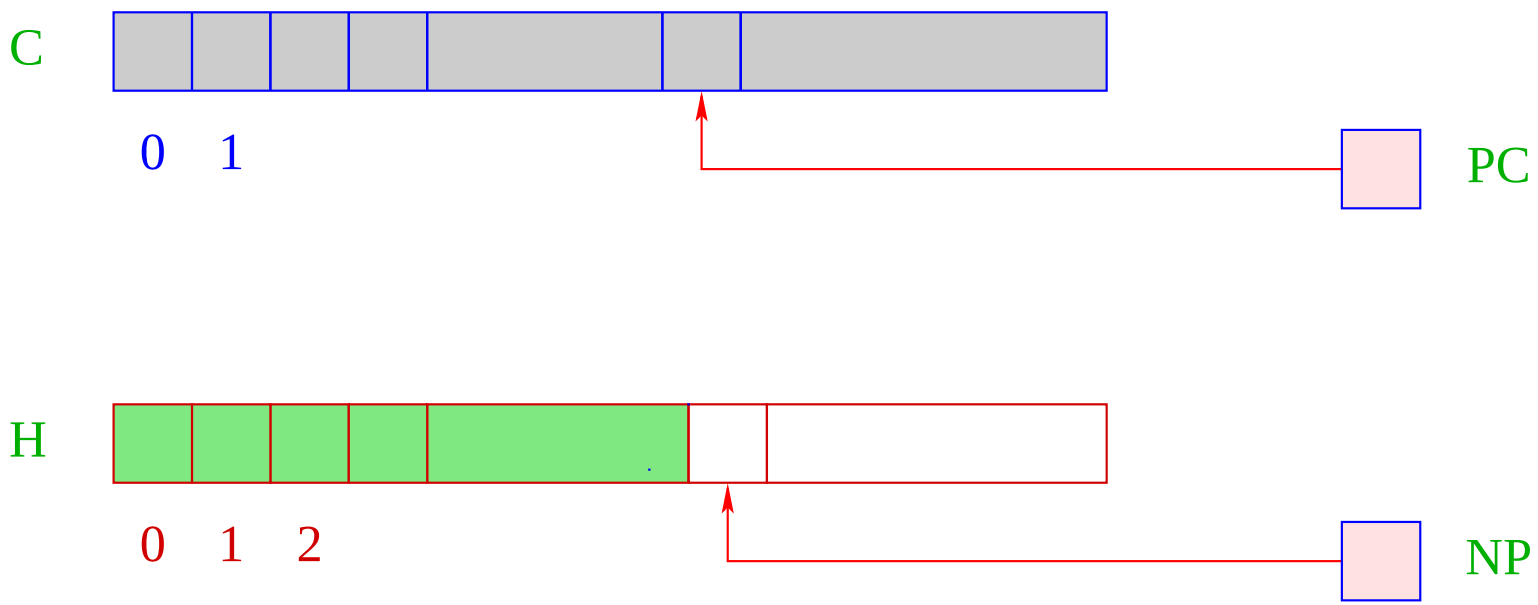
We extend C by a simple thread concept. In particular, we provide functions for:

- generating new threads: create();
- terminating a thread: exit();
- waiting for termination of a thread: join();
- mutual exclusion: lock(), unlock(); ...

In order to enable a parallel program execution, we extend the abstract machine (what else? :-)

40 Storage Organization

All threads share the same common code store and heap:

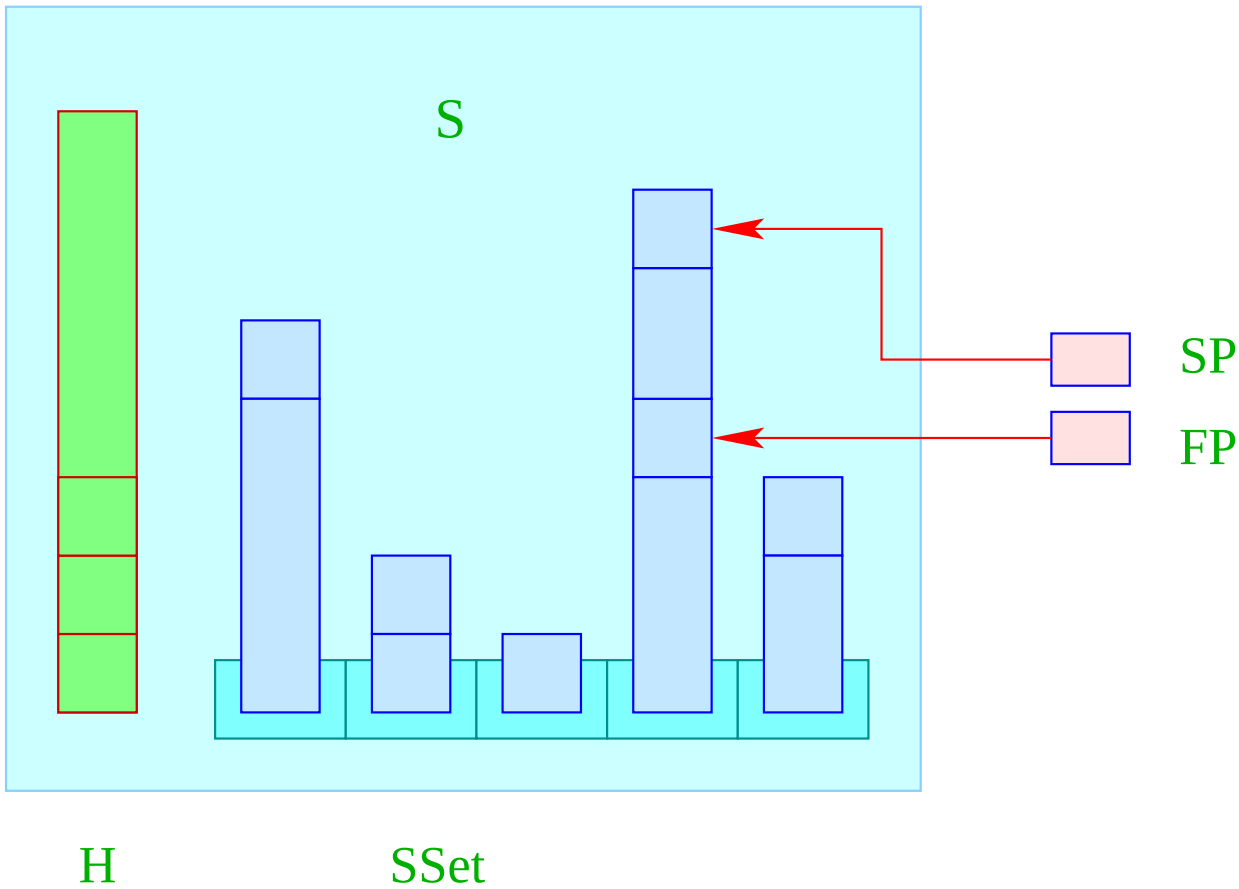


... similar to the CMa, we have:

C	=	Code Store – contains the CMa program;
		every cell contains one instruction;
PC	=	Program-Counter – points to the next executable instruction;
H	=	Heap –
		every cell may contain a base value or an address;
		the globals are stored at the bottom;
NP	=	New-Pointer – points to the first free cell.

For a simplification, we assume that the heap is stored in a separate segment.
The function malloc() then fails whenever NP exceeds the topmost border.

Every thread on the other hand needs its own stack:



In constrast to the CMa, we have:

SSet	=	Set of Stacks – contains the stacks of the threads;
		every cell may contain a base value of an address;
S	=	common address space for heap and the stacks;
SP	=	Stack-Pointer – points to the current topmost ocupied stack cell;
FP	=	Frame-Pointer – points to the current stack frame.

Warning:

- If all references pointed into the heap, we could use separate address spaces for each stack.

  Besides SP and FP, we would have to record the number of the current stack :-)

- In the case of C, though, we must assume that all storage reagions live within the same address space — only at different locations :-)

  SP Und FP then uniquely identify storage locations.

- For simplicity, we omit the extreme-pointer EP.

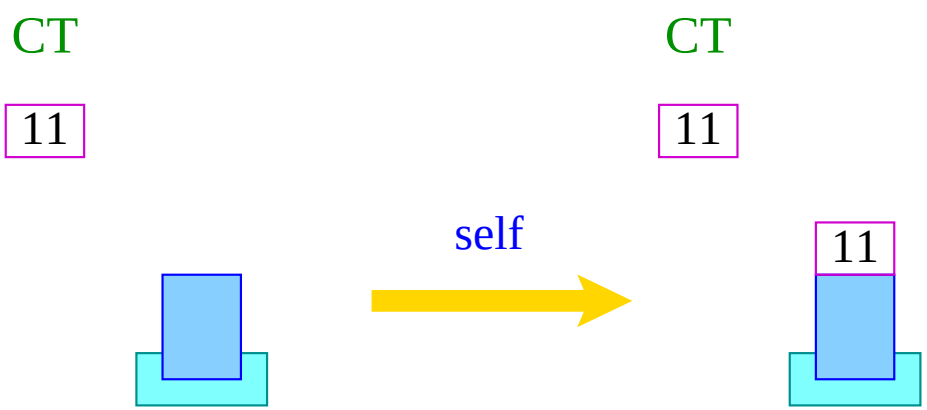
41 The Ready-Queue

Idea:

- Every thread has a unique number tid.
- A table TTab allows to determine for every tid the corresponding thread.
- At every point in time, there can be several executable threads, but only one running thread (per processor :-)
- the tid of the currently running thread is cept in the register CT (Current Thread).
- The function: **tid self ()** returns the tid of the current thread. Accordingly:

$$\text{code}_R\ \textbf{self}\ ()\ \rho \quad = \quad \text{self}$$

... where the instruction self pushes the content of the register CT onto the (current) stack:

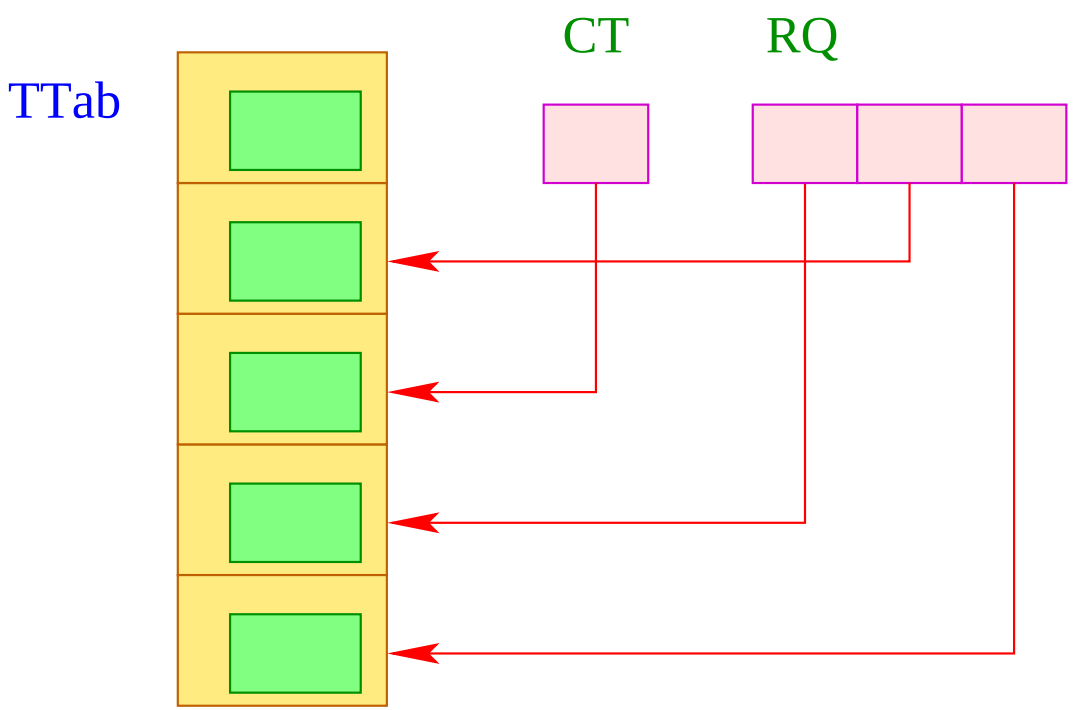


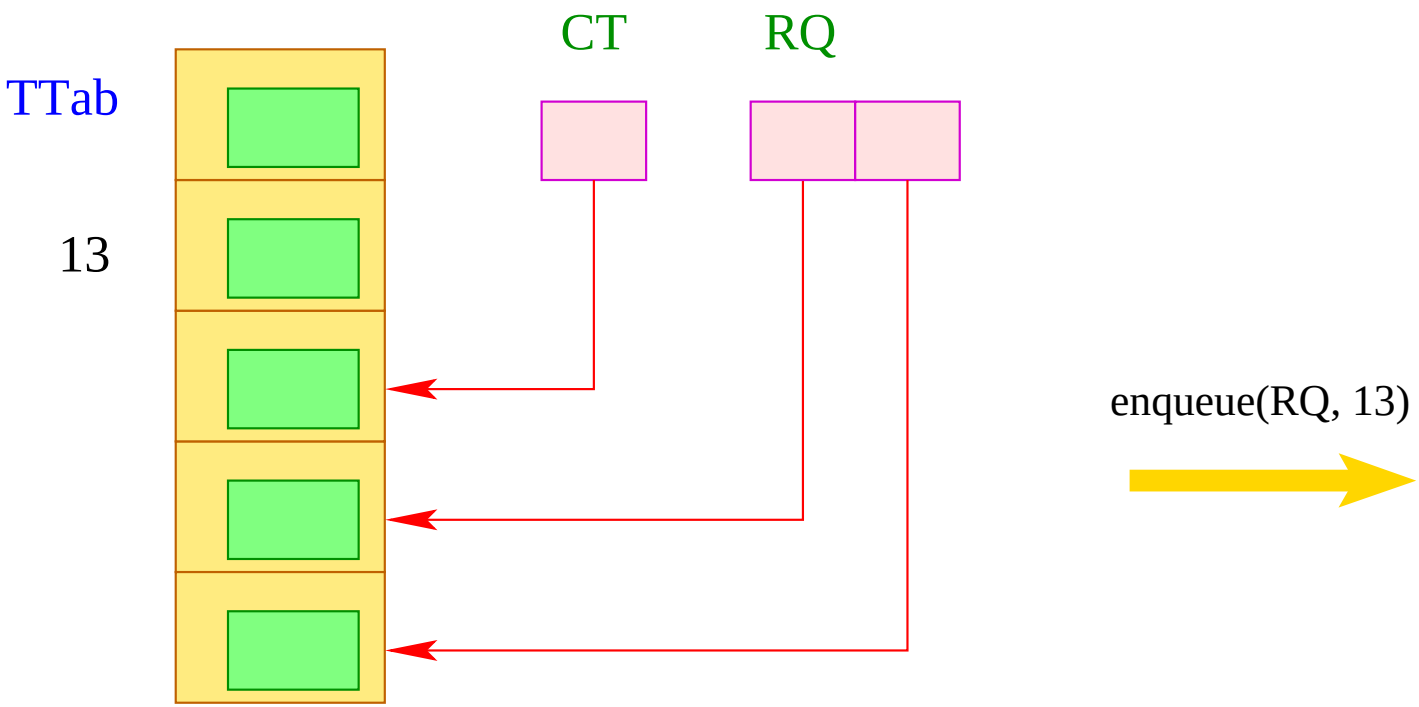
S[SP++] = CT;

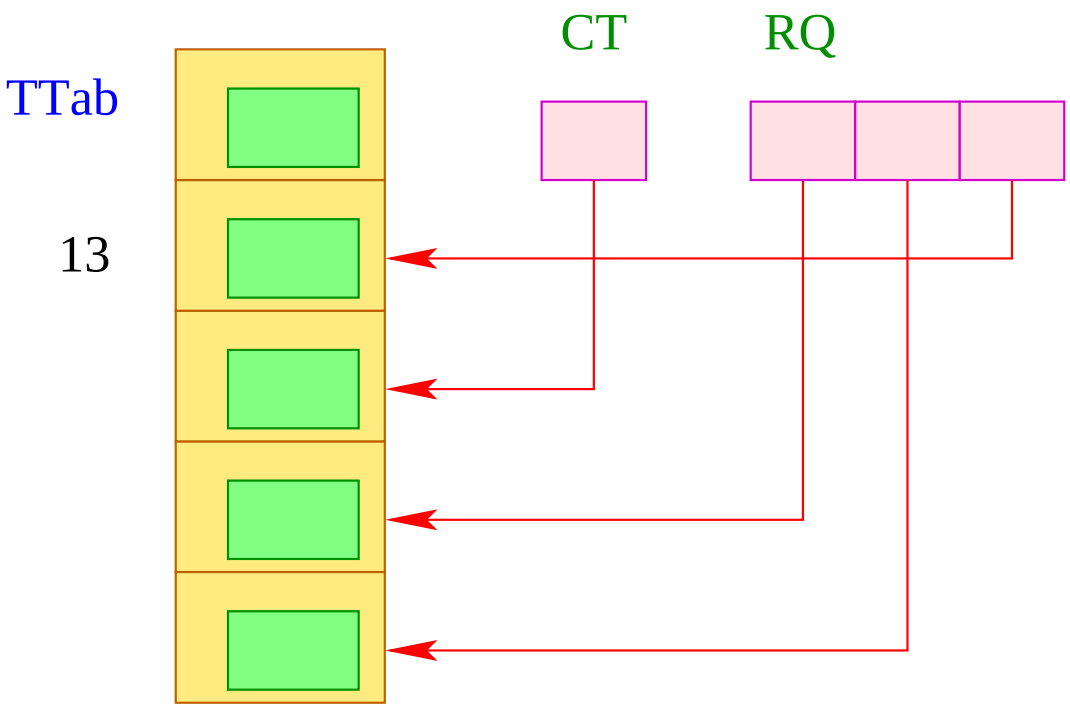
- The remaining executable threads (more precisely, their tid's) are maintained in the queue RQ (Ready-Queue).
- For queues, we need the functions:

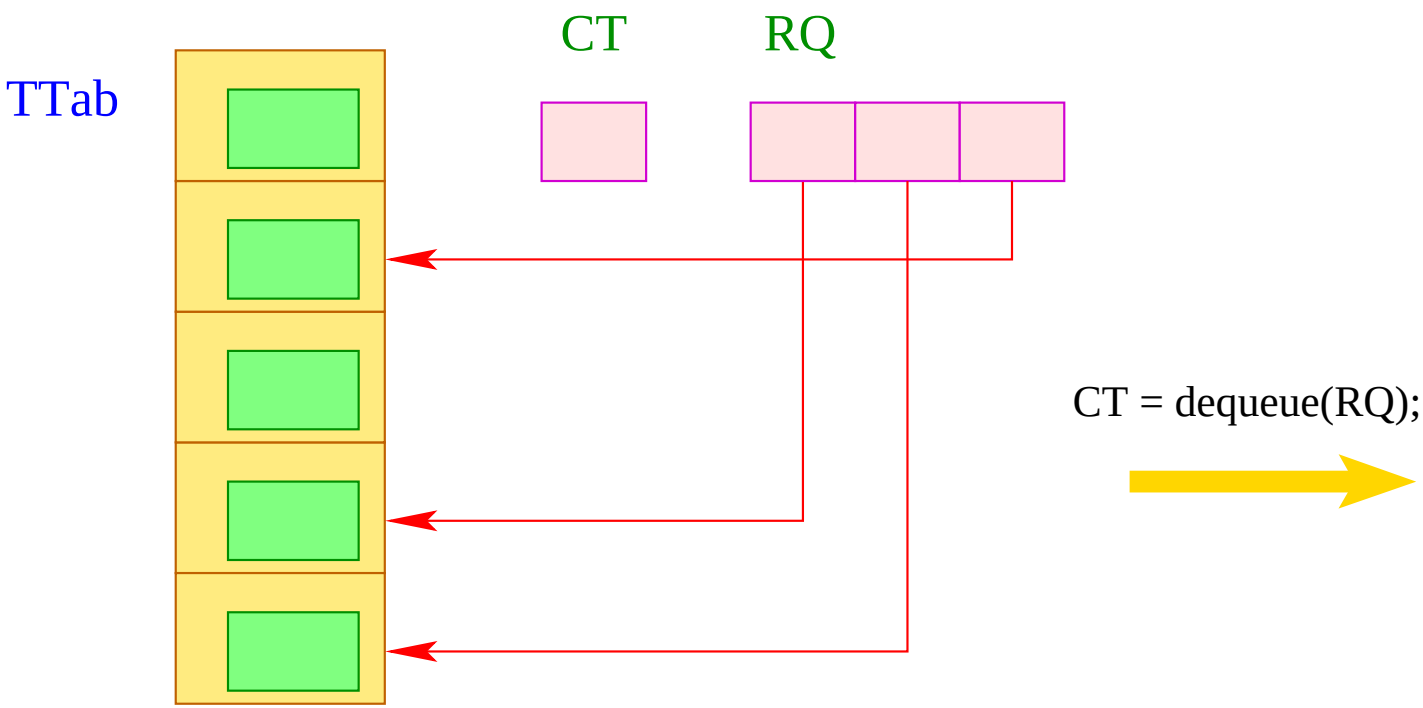
```
void enqueue (queue q, tid t),
tid dequeue (queue q)
```

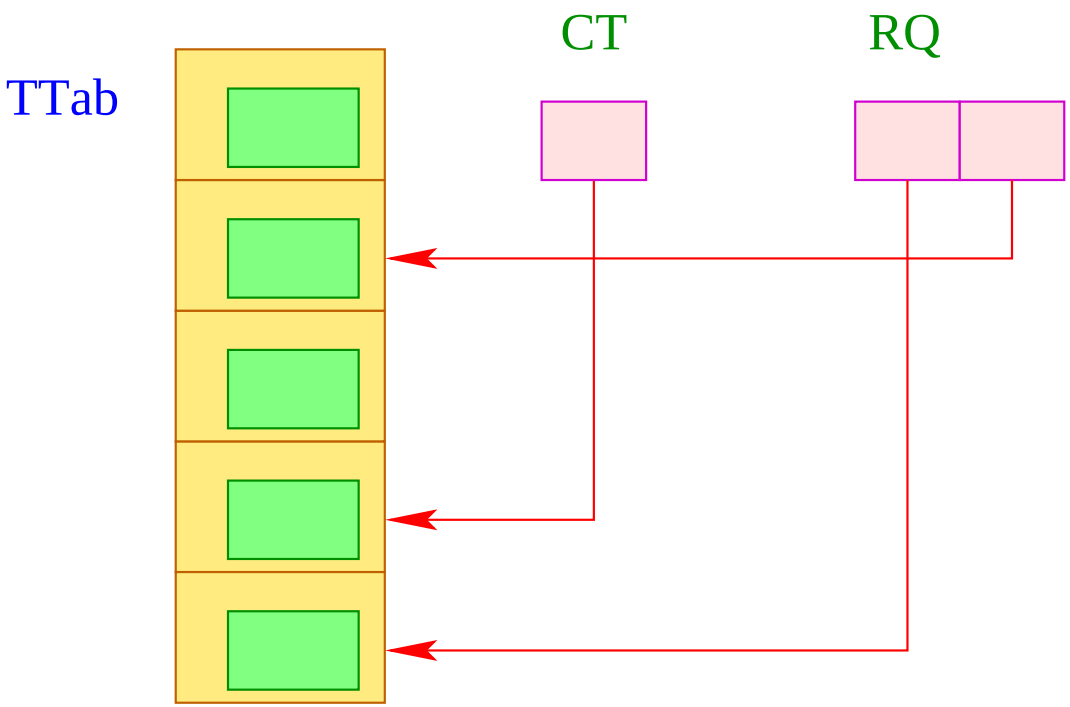
which insert a tid into a queue and return the first one, respectively ...


TTab
CT
RQ


CT
RQ
TTab
13
enqueue(RQ, 13)


CT
RQ
TTab
13


TTab
CT
RQ
CT = dequeue(RQ);


TTab
CT
RQ

If a call to dequeue () failed, it returns a value < 0 :-)

The thread table must contain for every thread, all information which is needed for its execution. In particular it consists of the registers PC, SP und FP:

Interrupting the current thread therefore requires to save these registers:

```
void save () {
     TTab[CT][0] = FP;
     TTab[CT][1] = PC;
     TTab[CT][2] = SP;
     }
```

Analogously, we restore these registers by calling the function:

```
void restore () {
     FP = TTab[CT][0];
     PC = TTab[CT][1];
     SP = TTab[CT][2];
     }
```

Thus, we can realize an instruction yield which causes a thread-switch:

```
tid ct = dequeue ( RQ );
if (ct ≥ 0) {
        save (); enqueue ( RQ, CT );
        CT = ct;
        restore ();
        }
```

Only if the ready-queue is non-empty, the current thread is replaced :-)

42 Switching between Threads

Problem:

We want to give each executable thread a fair chance to be completed.

$$\Longrightarrow$$

- Every thread must former or later be scheduled for running.
- Every thread must former or later be interrupted.

Possible Strategies:

- Thread switch only at explicit calls to a function `yield()` :-(
- Thread switch after every instruction $\Longrightarrow$ too expensive :-(
- Thread switch after a fixed number of steps $\Longrightarrow$ we must install a counter and execute `yield` at dynamically chosen points :-(

We insert thread switches at selected program points ...

- at the beginning of function bodies;
- before every jump whose target does not exceed the current PC ...

$\Longrightarrow$ rare :-))

The modified scheme for loops $s \equiv \textbf{while}\ (e)\ s$ then yields:

$$
\begin{array}{lll}
\text{code}\ s\ \rho & = & \text{A}: \quad \text{code}_{\text{R}}\ e\ \rho \\
 & & \qquad\ \text{jumpz B} \\
 & & \qquad\ \text{code}\ s\ \rho \\
 & & \qquad\ \text{yield} \\
 & & \qquad\ \text{jump A} \\
 & & \text{B}: \quad \ldots
\end{array}
$$

Note:

- **If-then-else**-Statements do not necessarily contain thread switches.
- **do-while**-Loops require a thread switch at the end of the condition.
- Every loop should contain (at least) one thread switch :-)
- Loop-Unroling reduces the number of thread switches.
- At the translation of **switch**-statements, we created a jump table behind the code for the alternatives. Nonetheless, we can avoid thread switches here.
- At freely programmed uses of jumpi as well as jumpz we should also insert thread switches before the jump (or at the jump target).
- If we want to reduce the number of executed thread switches even further, we could switch threads, e.g., only at every 100th call of yield ...

43 Generating New Threads

We assume that the expression: $s \equiv$ **create** (e_0, e_1) first evaluates the expressions e_i to the values f, a and then creates a new thread which computes $f\,(a)$.

If thread creation fails, s returns the value -1.

Otherwise, s returns the new thread's tid.

Tasks of the Generated Code:

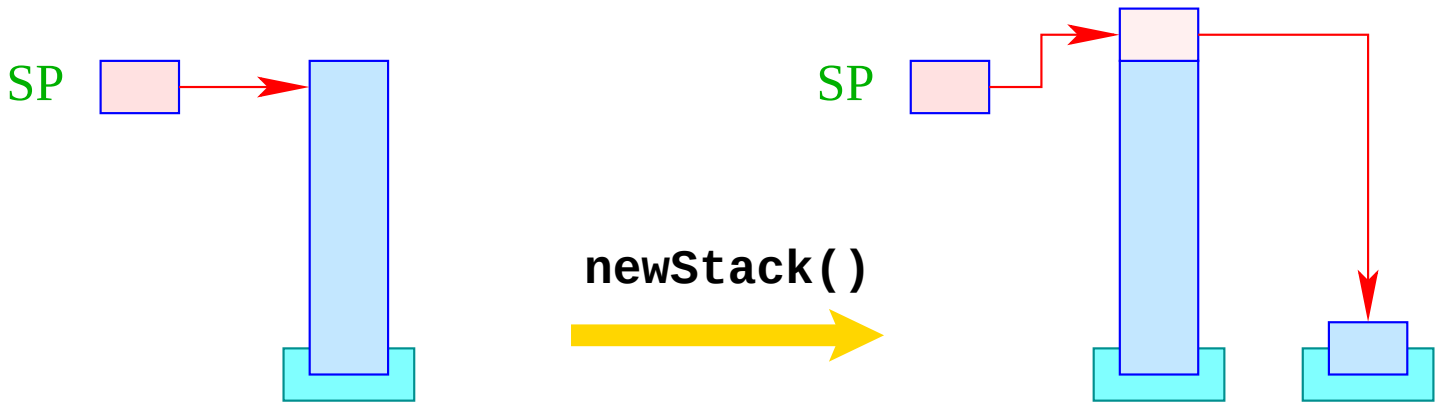
- Evaluation of the e_i;
- Allocation of a new run-time stack together with a stack frame for the evaluation of $f\,(a)$;
- Generation of a new tid;
- Allocation of a new entry in the TTab;
- Insertion of the new tid into the ready-queue.

The translation of s then is quite simple:

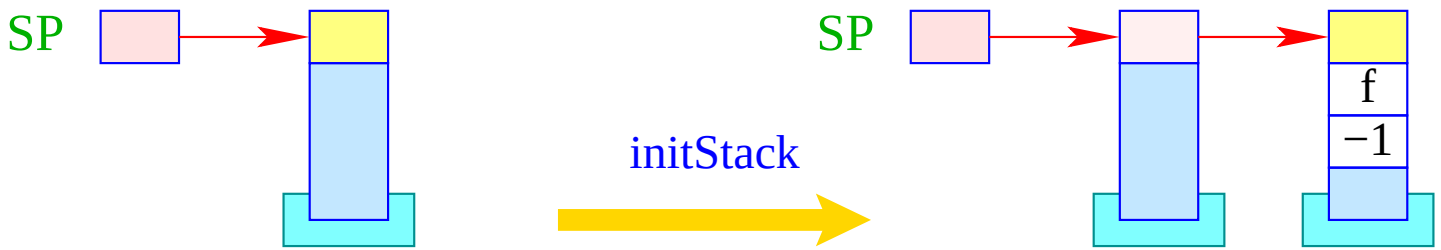
$$
\begin{array}{lcl}
\text{code}_\text{R}\ s\ \rho & = & \text{code}_\text{R}\ e_0\ \rho \\
 & & \text{code}_\text{R}\ e_1\ \rho \\
 & & \text{initStack} \\
 & & \text{initThread}
\end{array}
$$

where we assume the argument value occupies 1 cell :-)

For the implementation of initStack we need a run-time function newStack() which returns a pointer onto the first element of a new stack:



If the creation of a new stack fails, the value 0 is returned.




```
newStack();
if (S[SP]) {
        S[S[SP]+1] = -1;
        S[S[SP]+2] = f;
        S[S[SP]+3] = S[SP-1];
        S[SP-1] = S[SP]; SP--
    }
else S[SP = SP - 2] = -1;
```

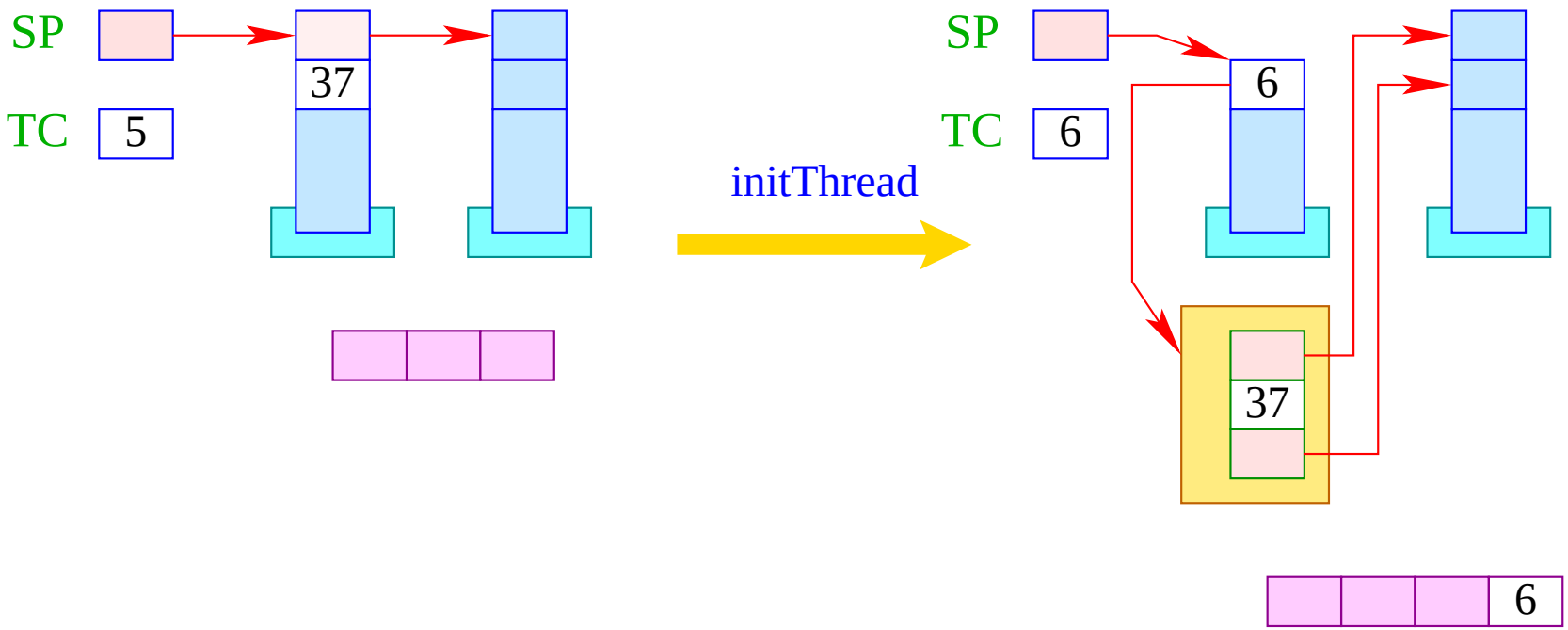
Note:

- The continuation address f points to the (fixed) code for the termination of threads.
- Inside the stack frame, we no longer allocate space for the EP $\Longrightarrow$ the return value has relative address -2.
- The bottom stack frame can be identified through FPold = -1 :-)

In order to create new thread ids, we introduce a new register TC (Thread Count).

Initially, TC has the value 0 (corresponds to the tid of the initial thread).

Before thread creation, TC is incremented by 1.


SP
TC
5
37
initThread
SP
TC
6
6
37
6

```
if (S[SP] ≥ 0) {
        tid = ++TCount;
        TTab[tid][0] = S[SP]-1;
        TTab[tid][1] = S[SP-1];
        TTab[tid][2] = S[SP];
        S[--SP] = tid;
        enqueue( RQ, tid );
}
```

44 Terminating Threads

Termination of a thread (usually :-) returns a value. There are two (regular) ways to terminate a thread:

1. The initial function call has terminated. Then the return value is the return value of the call.
2. The thread executes the statement **exit** (e); Then the return value equals the value of e.

Warning:

- We want to return the return value in the bottom stack cell.
- **exit** may occur arbitrarily deeply nested inside a recursion. Then we de-allocate all stack frames ...
- ... and jump to the terminal treatment of threads at address f .

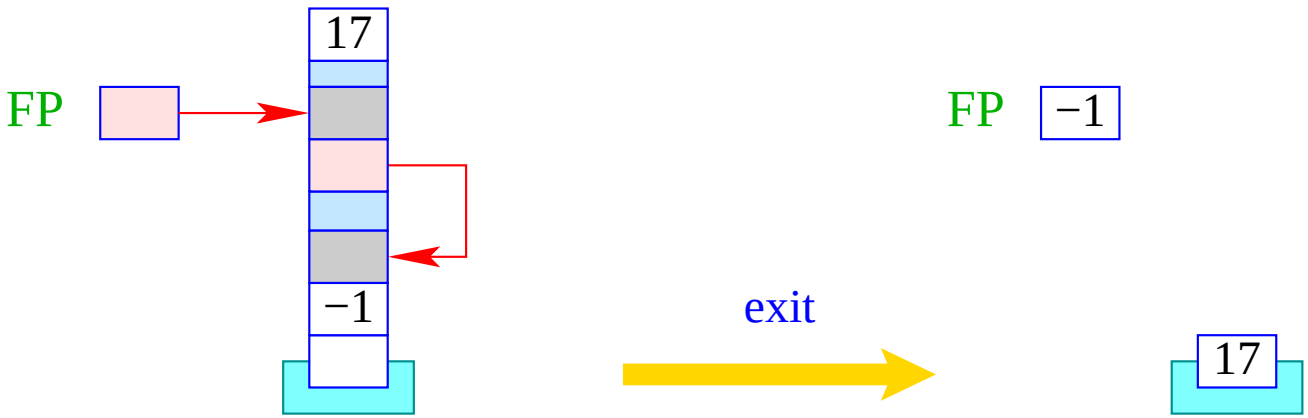
Therefore, we translate:

$$\begin{array}{lcl} \text{code } \textbf{exit}\ (e);\ \rho & = & \text{code}_{\text{R}}\ e\ \rho \\ & & \text{exit} \\ & & \text{term} \\ & & \text{next} \end{array}$$

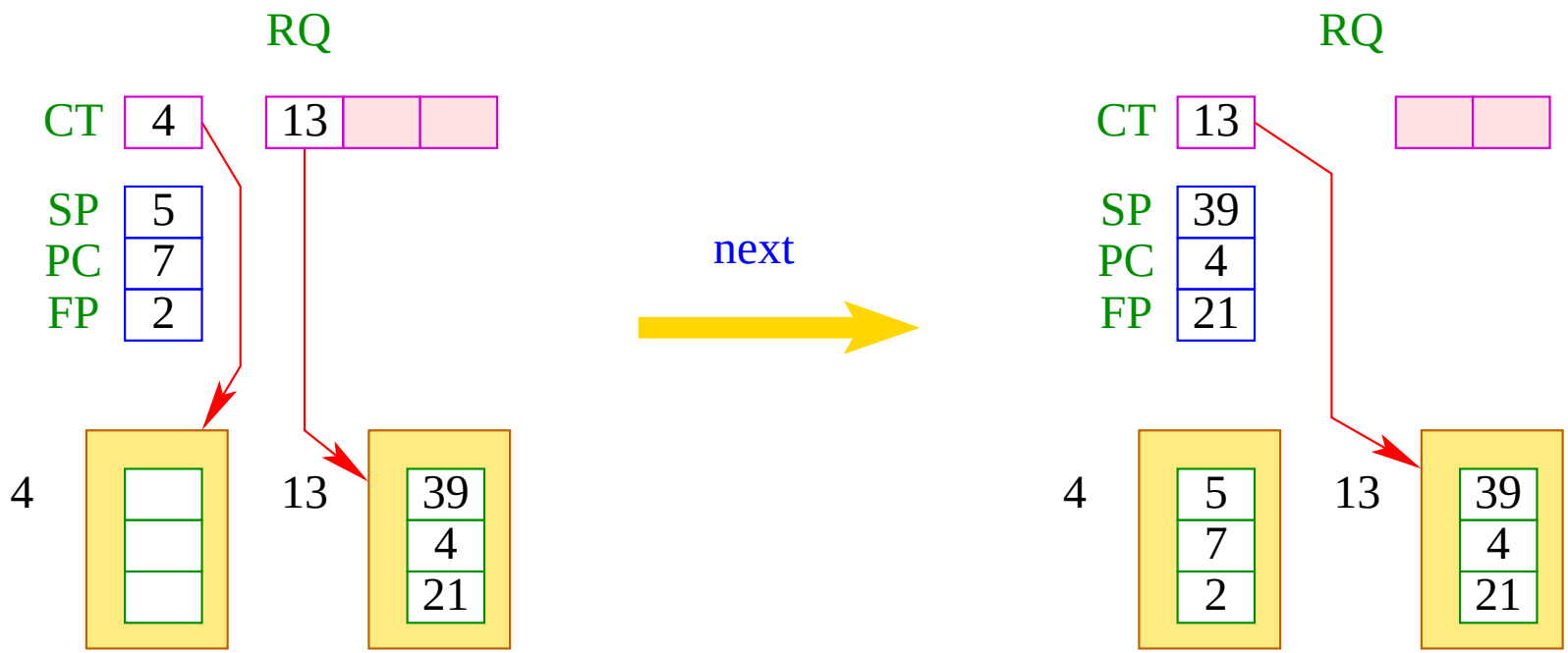
The instruction term is explained later :-)

The instruction exit successively pops all stack frames:

```
result = S[SP];
while (FP ≠ –1) {
        SP = FP–2;
        FP = S[FP–1];
        }
S[SP] = result;
```


17
FP
−1
exit
FP
−1
17

The instruction next activates the next executable thread:
in contrast to yield the current thread is not inserted into RQ .



Ist die Schlange RQ leer, wird zusätzlich If the queue RQ is empty, we additionally terminate the whole program:

```
if (0 > ct = dequeue( RQ ))  halt;
else {
        save ();
        CT = ct;
        restore ();
     }
```

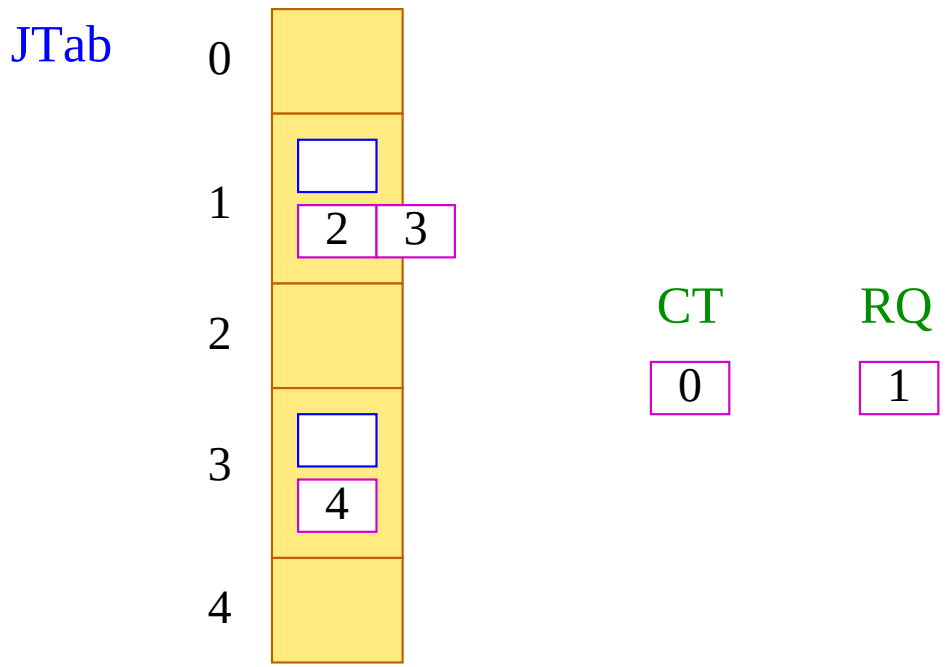
45 Waiting for Termination

Occaionally, a thread may only continue with its execution, if some other thread has terminated. For that, we have the expression **join** (e) where we assume that e evaluatges to a thread id tid.

- If the thread with the given tid is already terminated, we return its return value.
- If it is not yet terminated, we interrupt the current thread execution.
- We insert the current thread into the queue of treads already waiting for the termination.

  We save the current registers and switch to the next executable thread.
- Thread waiting for termination are maintained in the table JTab.
- There, we also store the return values of threads :-)

Example:



Thread 0 is running, thread 1 could run, threads 2 and 3 wait for the termination of 1, and thread 4 waits for the termination of 3.

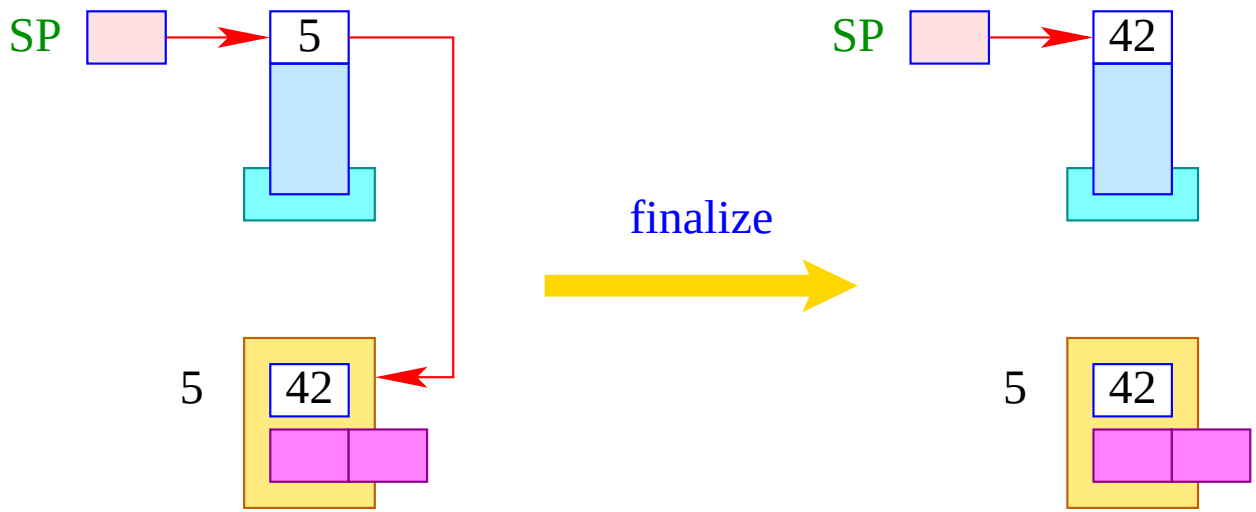
Thus, we translate:

$$\begin{array}{lcl} \text{code}_R\ \mathbf{join}\ (e)\ \rho & = & \text{code}_R\ e\ \rho \\ & & \text{join} \\ & & \text{finalize} \end{array}$$

... where the instruction join is defined by:

```
tid = S[SP];
if (TTab[tid][1] ≥ 0) {
        enqueue ( JTab[tid], CT );
        next
    }
```

... accordingly:



S[SP] = JTab[tid][1];

The instruction sequence:

```
term
next
```

is executed before a thread is terminated.
Therefore, we store them at the location f.

The instruction next switches to the next executable thread. Before that, though,

- ... the last stack frame must be popped and the result be stored in the table JTab ;
- ... the thread must be marked as terminated, e.g., by additionally setting the PC to -1;
- ... all threads must be notified which have waited for the termination.

For the instruction term this means:

```
PC = –1;
JTab[CT][1] = S[SP];
freeStack(SP);
while (0 ≤ tid = dequeue ( JTab[CT][0] ))
        enqueue ( RQ, tid );
```

The run-time function `freeStack (int adr)` removes the (one-element) stack at the location adr :

46 Mutual Exclusion

A mutex is an (abstract) datatype (in the heap) which should allow the programmer to dedicate exclusive access to a shared resource (mutual exclusion).

The datatype supports the following operations:

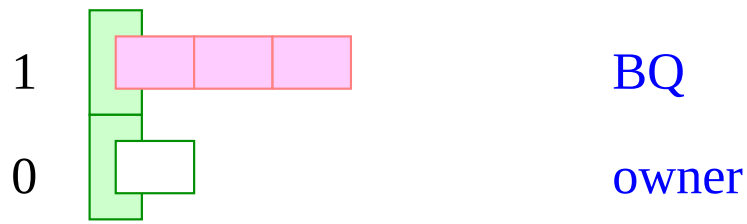
Mutex $*$ newMutex ();	—	creates a new mutex;
void lock (**Mutex** $*$me);	—	tries to acquire the mutex;
void unlock (**Mutex** $*$me);	—	releases the mutex;

Warning:

A thread is only allowed to release a mutex if it has owned it beforehand :-)

A mutex me consists of:

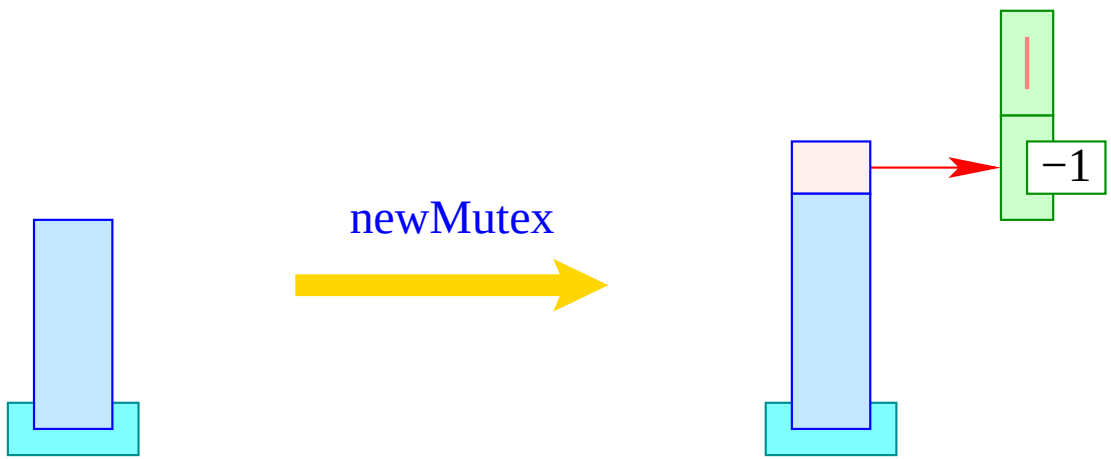
- the tid of the current owner (or -1 if there is no one);
- the queue BQ of blocked threads which want to acquire the mutex.



Then we translate:

$$\text{code}_R\ \textbf{newMutex}\ ()\ \rho \quad = \quad \text{newMutex}$$

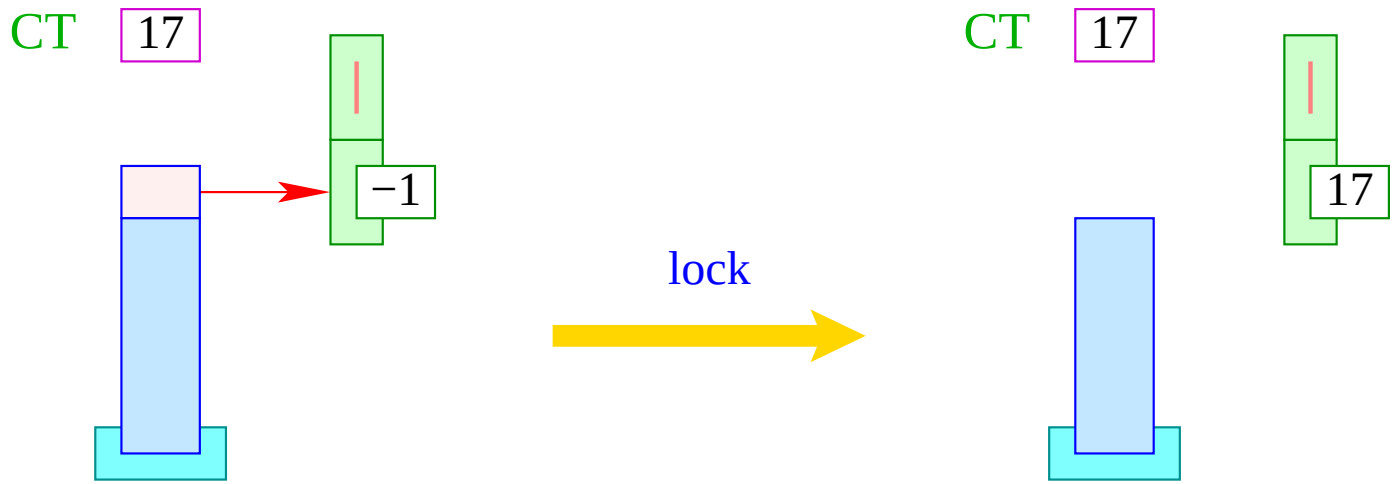
where:



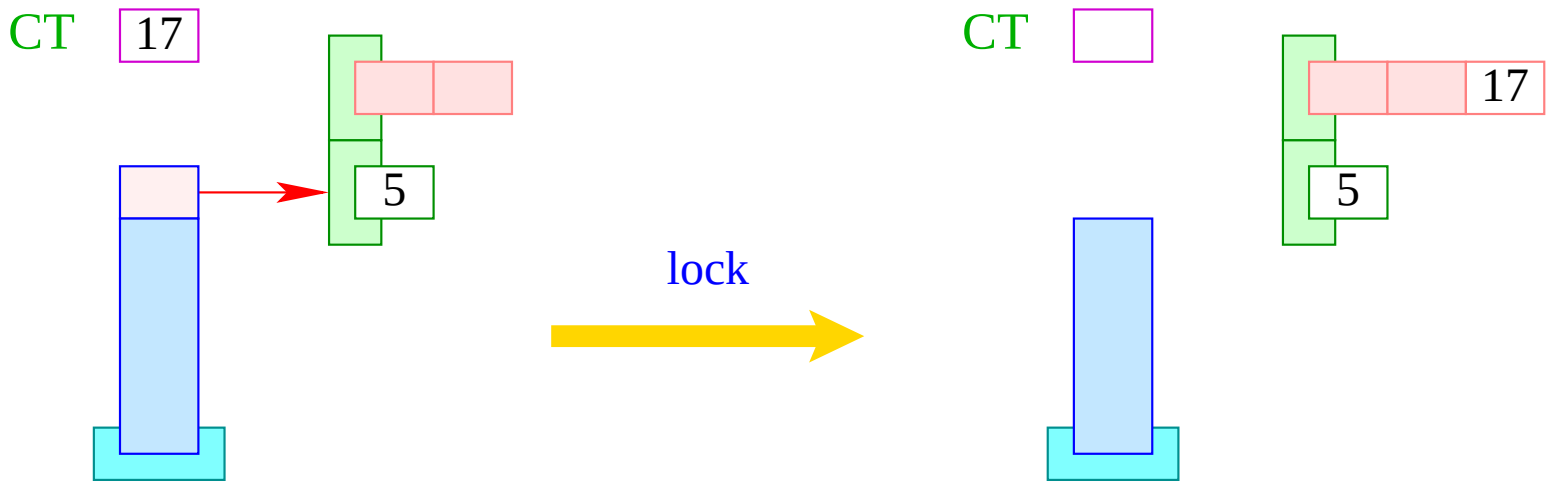
Then we translate:

$$\text{code } \textbf{lock } (e);\ \rho \quad = \quad \begin{array}{l} \text{code}_{\text{R}}\ e\ \rho \\ \text{lock} \end{array}$$

where:



If the mutex is already owned by someone, the current thread is interrupted:

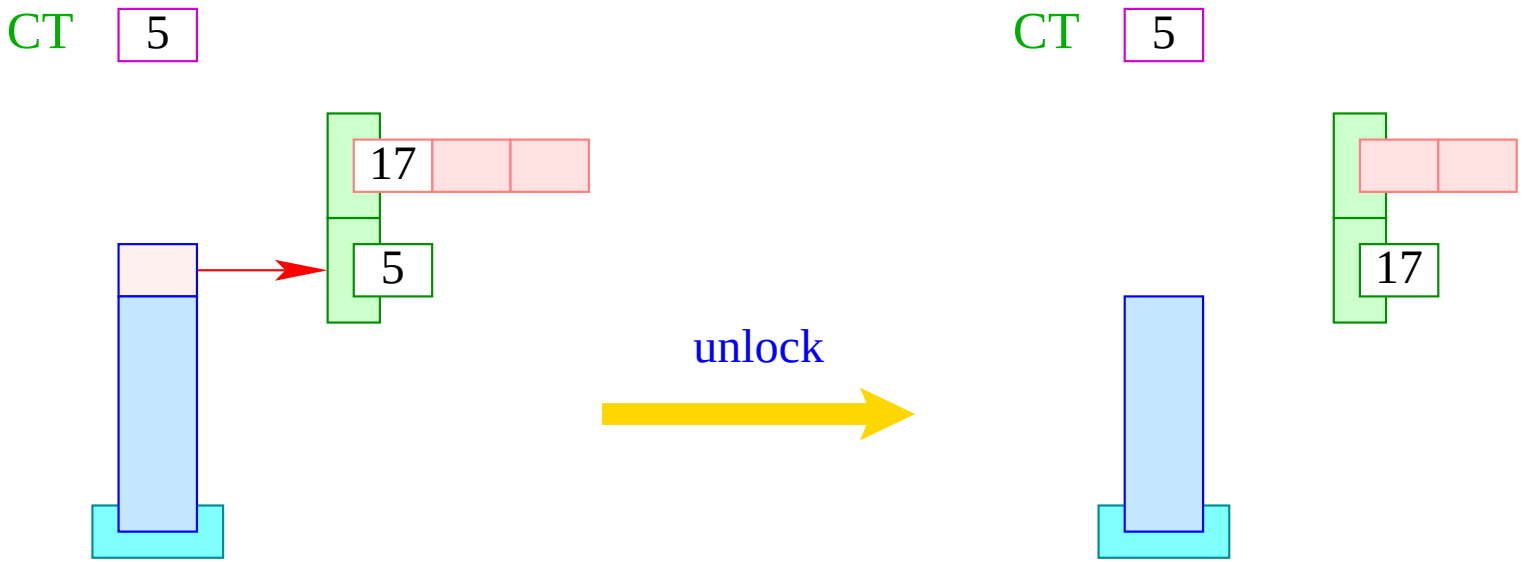



```
if (S[S[SP]] < 0)    S[S[SP−−]] = CT;
else {
        enqueue ( S[SP−−]+1, CT );
        next;
    }
```

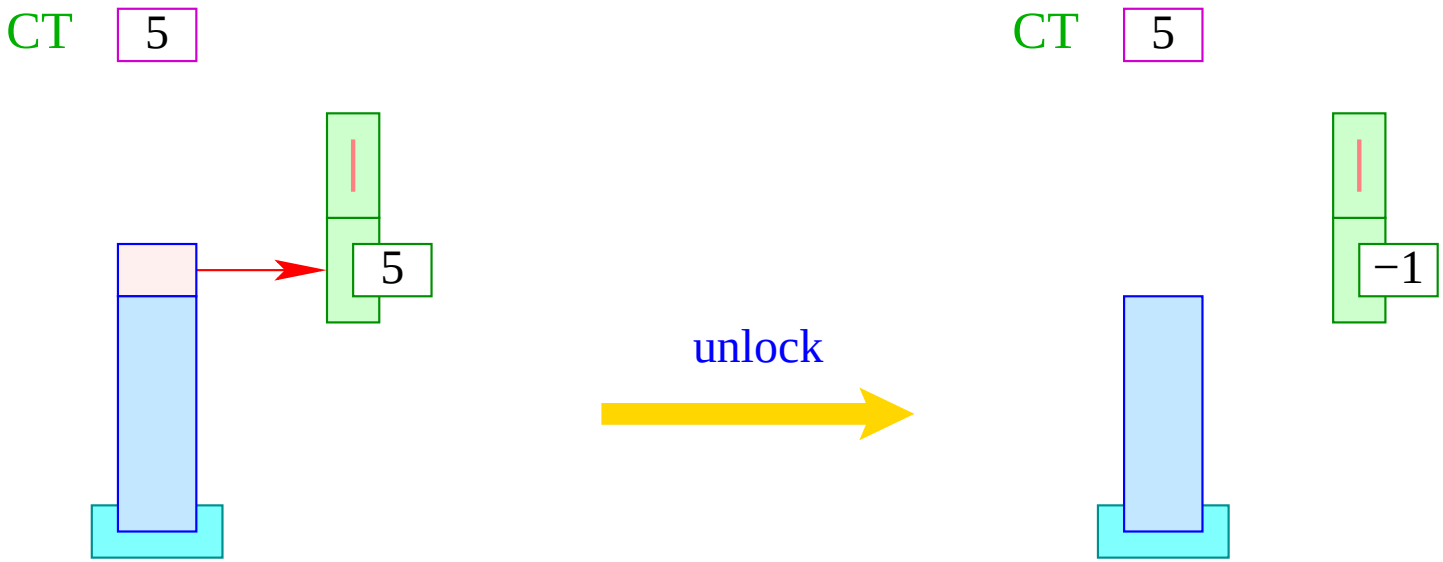
Accordingly, we translate:

$$\begin{array}{lcl} \text{code } \mathbf{unlock}\ (e);\ \rho & = & \text{code}_{\text{R}}\ e\ \rho \\ & & \text{unlock} \end{array}$$

where:



If the queue BQ is empty, we release the mutex:




```
if (S[S[SP]] ≠ CT)    Error ("Illegal unlock!");
if (0 > tid = dequeue ( S[SP]+1))    S[S[SP– –]] = –1;
else {
        S[S[SP--]] = tid;
        enqueue ( RQ, tid );
    }
```

47 Waiting for Better Wheather

It may happen that a thread owns a mutex but must wait until some extra condition is true.

Then we want the thread to remain in-active until it is told otherwise.

For that, we use condition variables. A condition variable consists of a queue WQ of waiting threads :-)

For condition variables, we introduce the functions:

CondVar $*$ newCondVar ();	—	creates a new condition variable;
void wait (**CondVar** $*$ cv), **Mutex** $*$ me);	—	enqueues the current thread;
void signal (**CondVar** $*$ cv);	—	re-animates one waiting thread;
void broadcast (**CondVar** $*$ cv);	—	re-animates all waiting threads.

Then we translate:

$$\text{code}_{\text{R}}\ \textbf{newCondVar}\ ()\ \rho \quad = \quad \text{newCondVar}$$

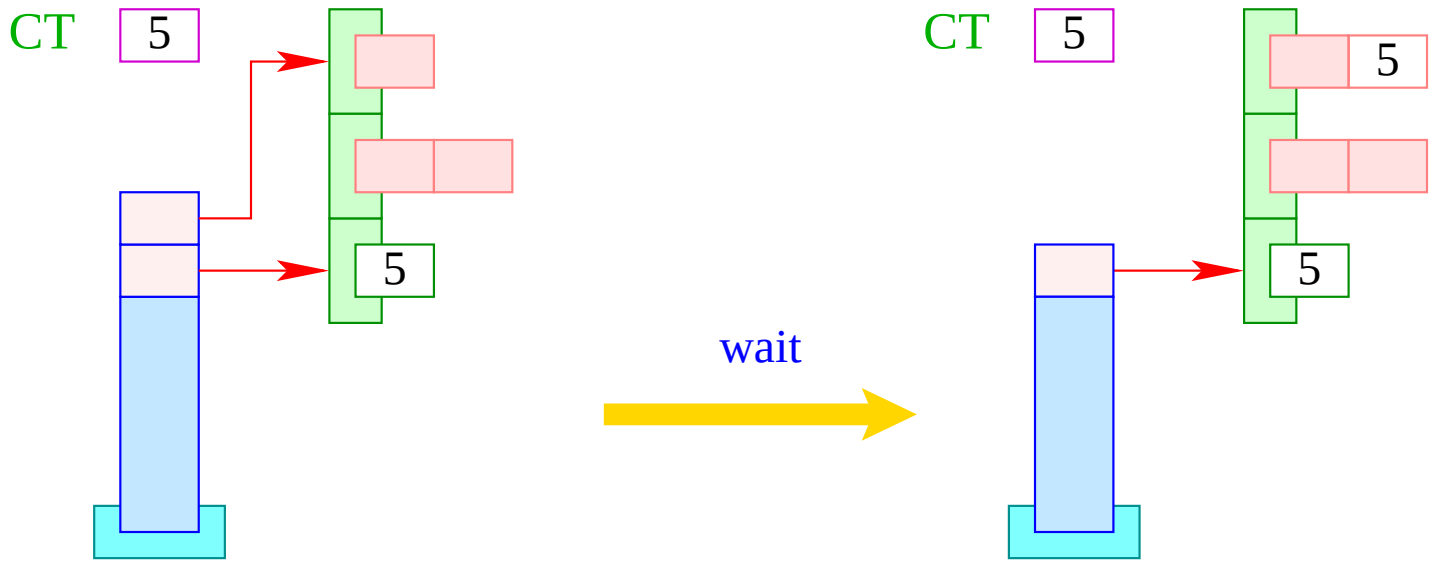
where:

After enqueuing the current thread, we release the mutex. After re-animation, though, we must acquire the mutex again.

Therefore, we translate:

$$
\begin{aligned}
\text{code } \textbf{wait } (e_0, e_1);\ \rho \quad = \quad & \text{code}_\text{R}\ e_1\ \rho \\
& \text{code}_\text{R}\ e_0\ \rho \\
& \text{wait} \\
& \text{dup} \\
& \text{unlock} \\
& \text{next} \\
& \text{lock}
\end{aligned}
$$

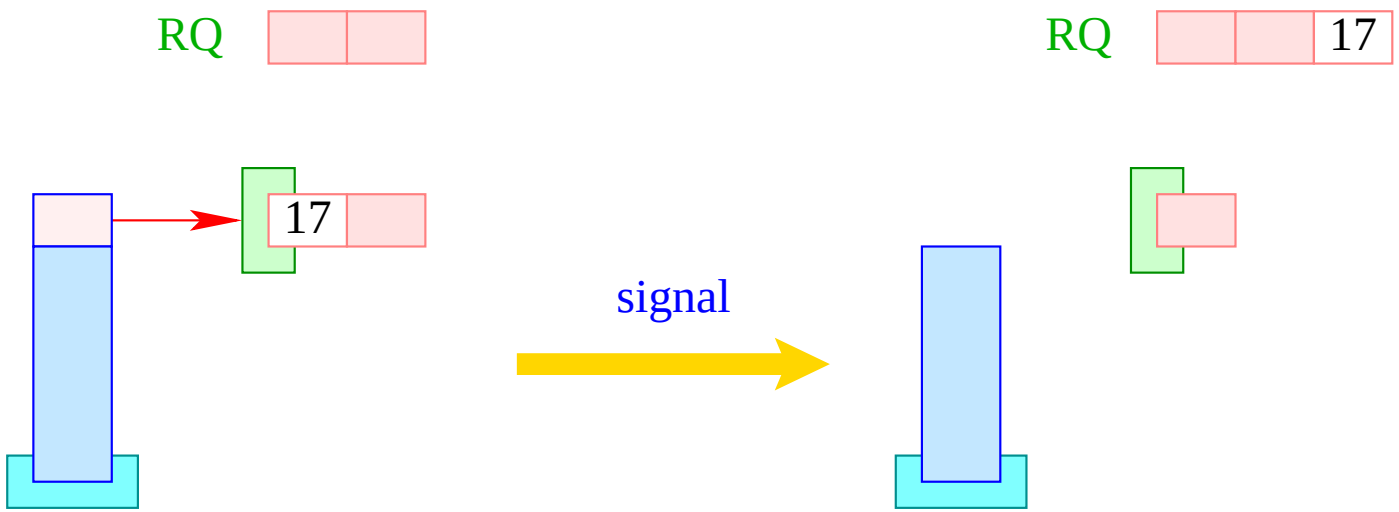
where ...



if (S[S[SP-1]] $\neq$ CT) Error ("Illegal wait!");
enqueue (S[SP], CT); SP--;

Accordingly, we translate:

$$\text{code } \textbf{signal } (e);\ \rho \quad = \quad \text{code}_{\text{R}}\ e\ \rho$$
$$\text{signal}$$




```
if (0 ≤ tid = dequeue ( S[SP]))
        enqueue ( RQ, tid );
SP--;
```

Analogously:

$$\text{code } \textbf{broadcast}\ (e);\ \rho \quad = \quad \text{code}_R\ e\ \rho \\ \text{broadcast}$$

where the instruction broadcast enqueues all threads from the queue WQ into the ready-queue RQ :

```
while (0 ≤ tid = dequeue ( S[SP]))
        enqueue ( RQ, tid );
SP--;
```

Warning:

The re-animated threads are not blocked !!!

When they become running, though, they first have to acquire their mutex :-)

48 Example: Semaphores

A semaphore is an abstract datatype which controls the access of a bounded number of (identical) resources.

Operations:

Sema $*$ newSema (**int** n)	—	creates a new semaphore;
void Up (Sema $*$ s)	—	increases the number of free resources;
void Down (Sema $*$ s)	—	decreases the number of available resources.

Therefore, a semaphore consists of:

- a counter of type **int**;
- a mutex for synchronizing the semaphore operations;
- a condition variable.

```
typedef struct {
        Mutex * me;
        CondVar * cv;
        int count;
    } Sema;
```

```
Sema * newSema (int n) {
        Sema * s;
        s = (Sema *) malloc (sizeof (Sema));
        s→me = newMutex ();
        s→cv = newCondVar ();
        s→count = n;
        return (s);
    }
```

The translation of the body amounts to:

```
alloc 1
loadc 3
new
storer 2
pop
```

```
newMutex
loadr 2
store
pop
```

```
newCondVar
loadr 2
loadr 2
loadc 1
add
store
pop
```

```
loadr 1
loadr 2
loadc 2
add
store
pop
```

```
loadr 2
storer -2
return
```

The function Down() decrements the counter.

If the counter becomes negative, **wait** is called:

```
void Down (Sema * s) {
        Mutex *me;
        me = s→me;
        lock (me);
        s→count– –;
        if (s→count < 0)   wait (s→cv,me);
        unlock (me);
    }
```

The translation of the body amounts to:

```
alloc 1          loadc 2          add              loadc 1
loadr 1          add              store            add
load             load             loadc 0          load
storer 2         loadc 1          le               wait
lock             sub              jumpz A    A:    loadr 2
                 loadr 1          loadr 2          unlock
loadr 1          loadc 2          loadr 1          return
```

The function Up() increments the counter again.

If it is afterwards not yet positive, there still must exist waiting threads. One of these is sent a signal:

```
void Up (Sema * s) {
        Mutex *me;
        me = s→me;
        lock (me);
        s→count++;
        if (s→count ≤ 0)   signal (s→cv);
        unlock (me);
    }
```

The translation of the body amounts to:

```
alloc 1          loadc 2          add              loadc 1
loadr 1          add              store            add
load             load             loadc 0          load
storer 2         loadc 1          le               signal
lock             add              jumpz A    A:    loadr 2
                 loadr 1                           unlock
loadr 1          loadc 2          loadr 1          return
```

49 Stack-Management

Problem:

- All threads live within the same storage.
- Every thread requires its own stack (at least conceptually).

1. Idea:

Allocate for each new thread a fixed amount of storage space.

Then we implement:

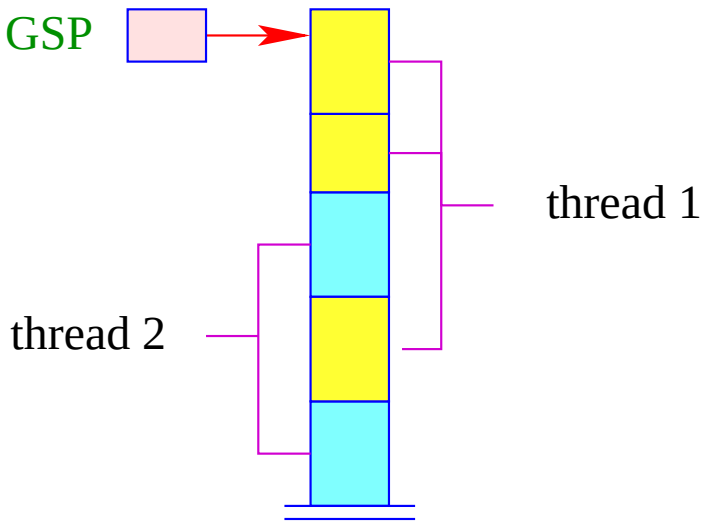
```
void *newStack() { return malloc(M); }
void freeStack(void *adr) { free(adr); }
```

Problem:

- Some threads consume much, some only little stack space.
- The necessary space is statically typically unknown :-(

2. Idea:

- Maintain all stacks in one joint Frame-Heap FH :-)
- Take care that the space inside the stack frame is sufficient at least for the current function call.
- A global stack-pointer GSP points to the overall topmost stack cell ...



Allocation and de-allocation of a stack frame makes use of the run-time functions:

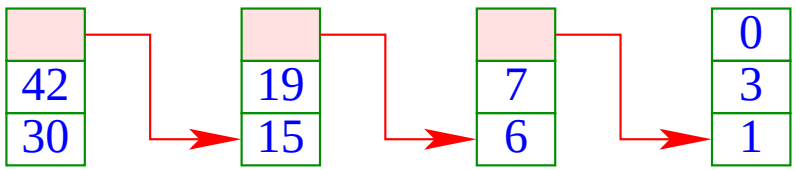
```
int newFrame(int size) {
    int result = GSP;
    GSP = GSP+size;
    return result;
    }

void freeFrame(int sp, int size);
```

Warning:

The de-allocated block may reside inside the stack :-(

$$\Longrightarrow$$

We maintain a list of freed stack blocks :-)

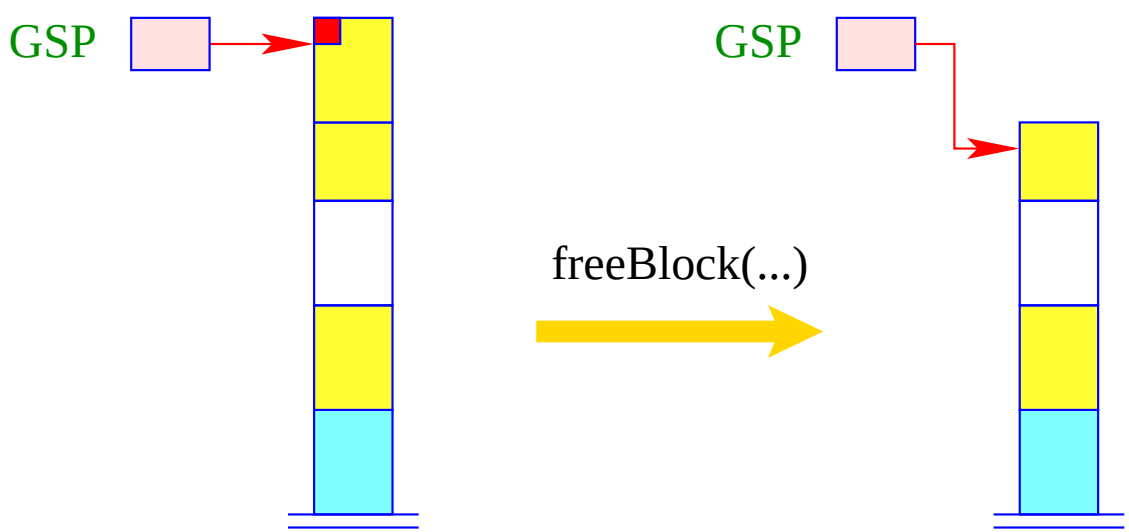


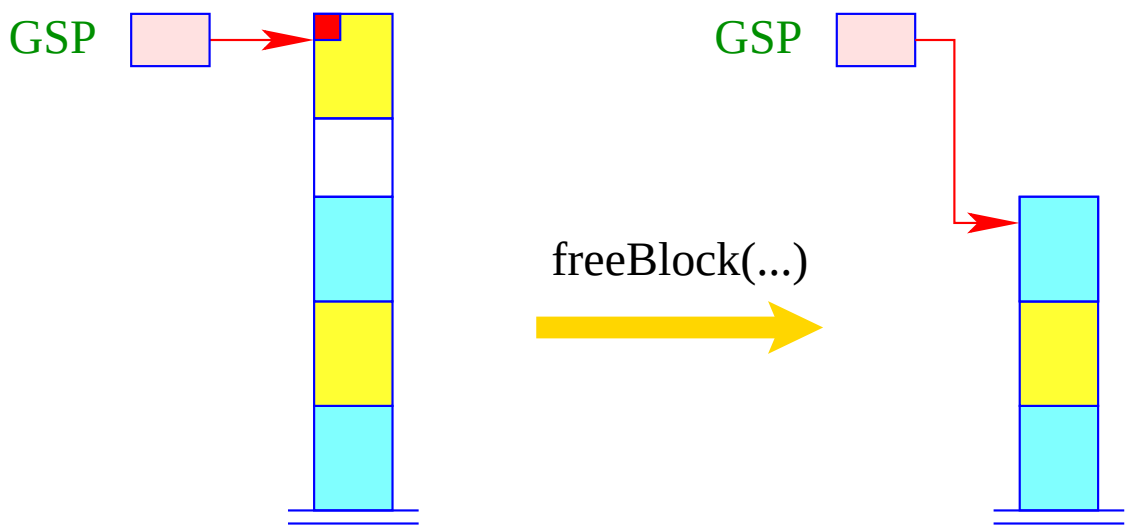
This list supports a function

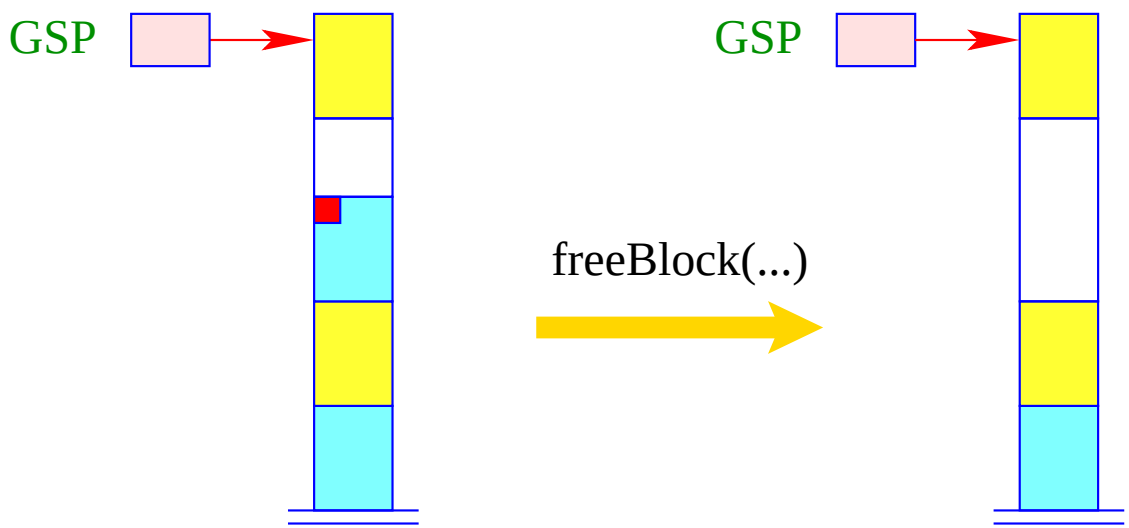
```
void insertBlock(int max, int min)
```

which allows to free single blocks.

- If the block is on top of the stack, we pop the stack immediately;
- ... together with the blocks below – given that these have already been marked as de-allocated.
- If the block is inside the stack, we merge it with neighbored free blocks:


GSP
freeBlock(...)
GSP


GSP
GSP
freeBlock(...)


GSP
freeBlock(...)
GSP

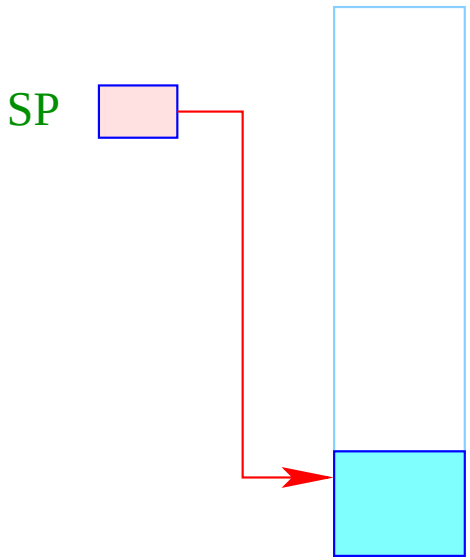
Approach:

We allocate a fresh block for every function call ...

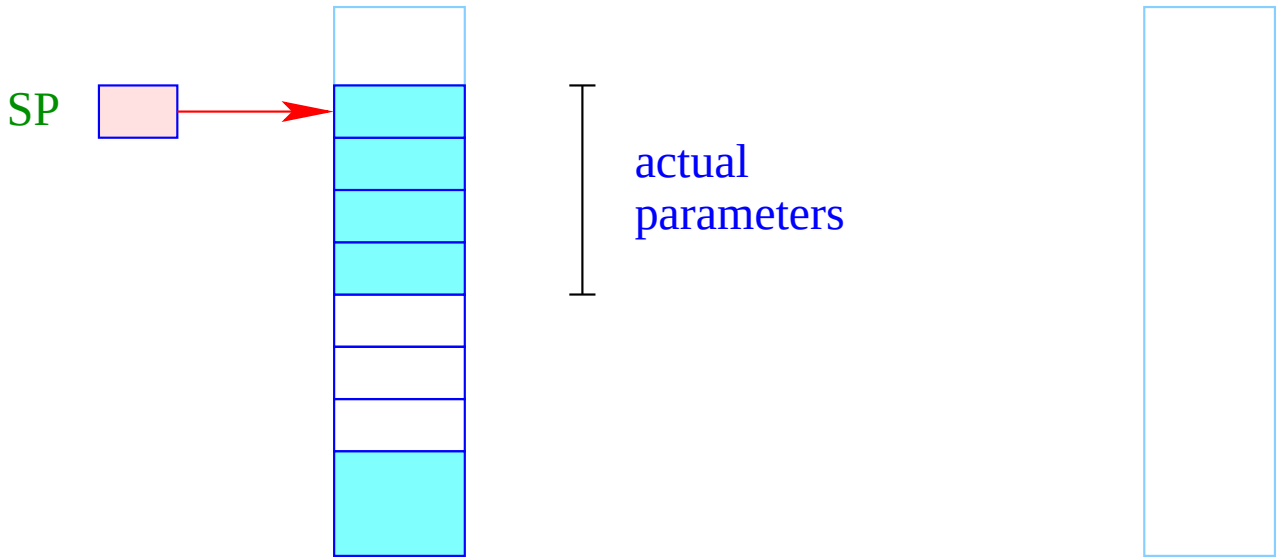
Problem:

When ordering the block before the call, we do not yet know the space consumption of the called function :-(

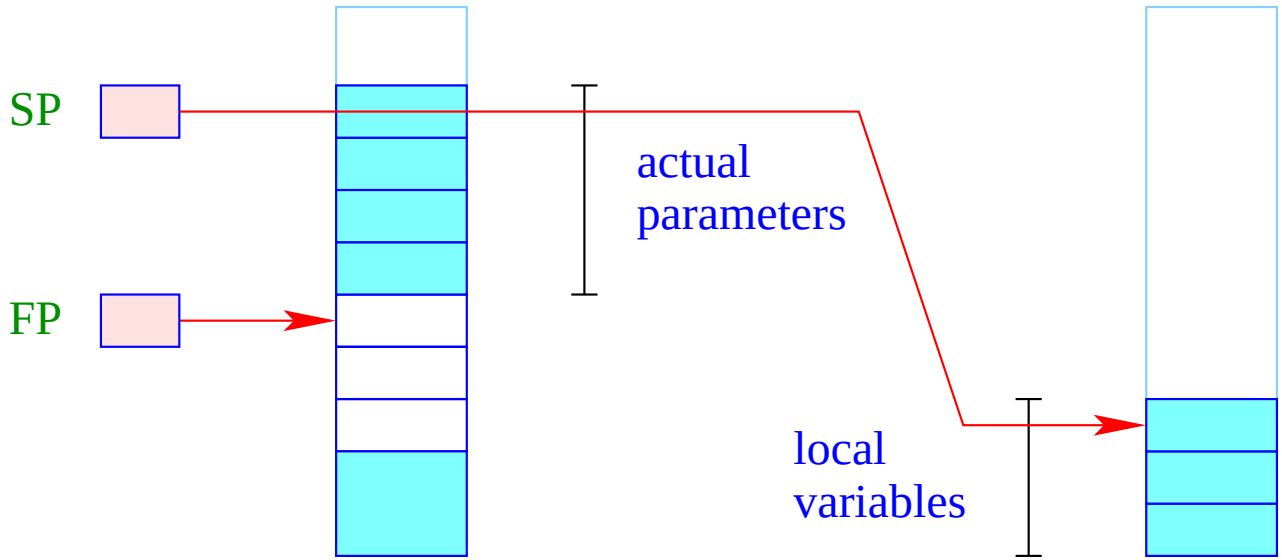
$\Longrightarrow$ We order the new block after entering the function body!



Organisational cells as well as actual parameters must be allocated inside the old block ...



When entering the new function, we now allocate the new block ...



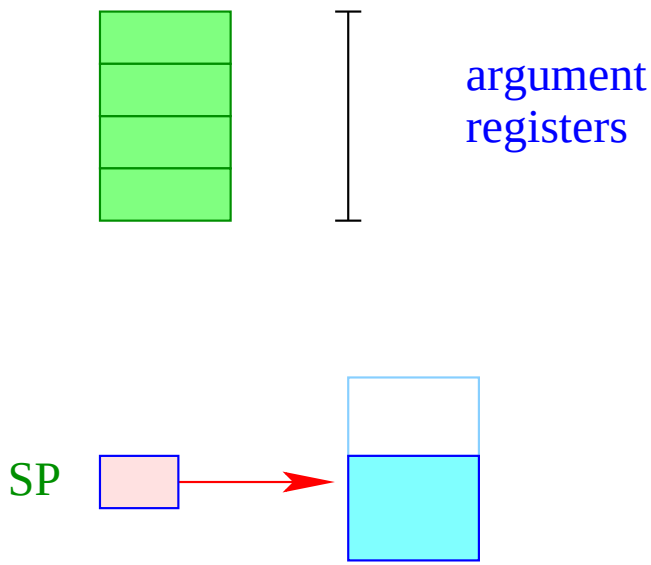
Inparticular, the local variables reside in the new block ...

$\Longrightarrow$ We address ...

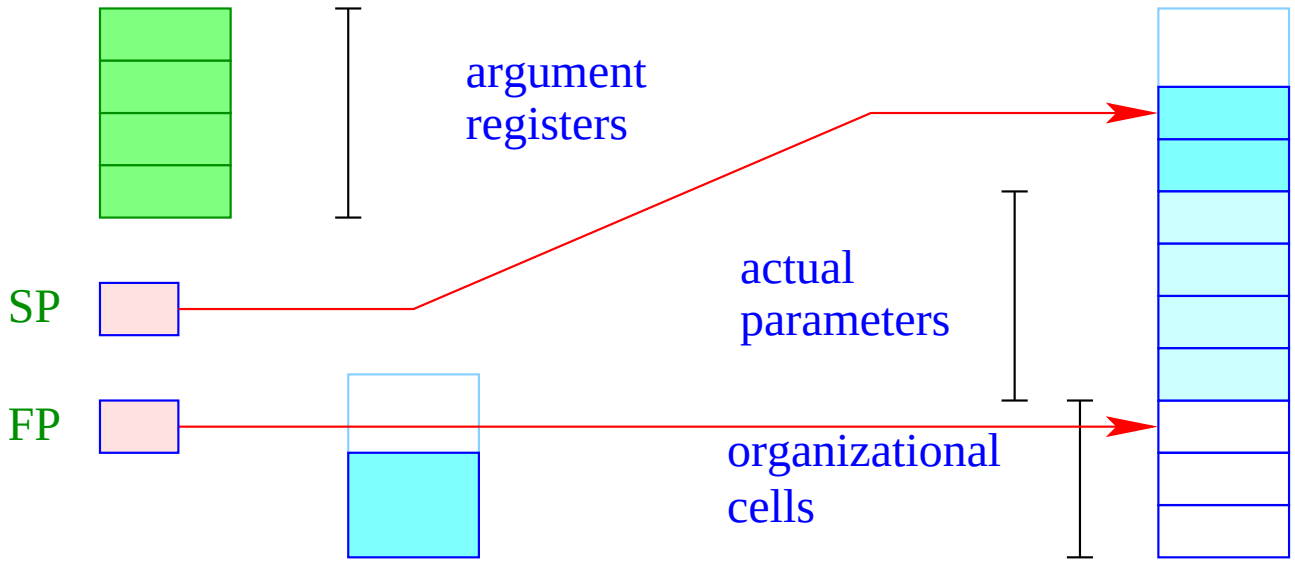
- the formal parameters relatively to the frame-pointer;
- the local variables relatively to the stack-pointer :-)

$\Longrightarrow$ We must re-organize the complete code generation ... :-(

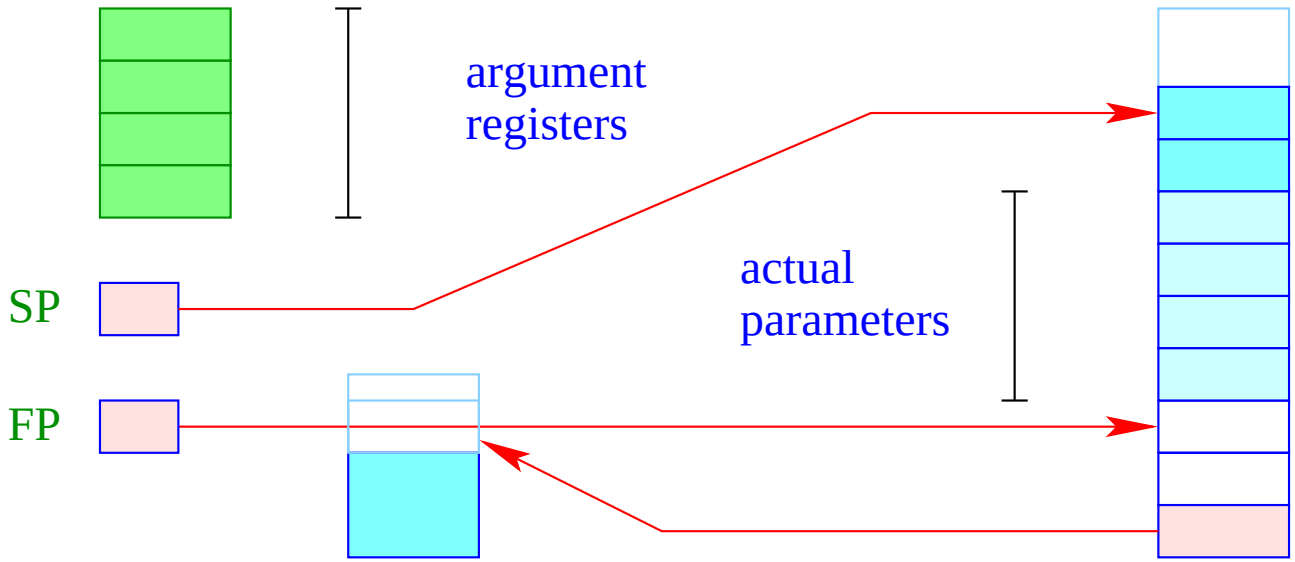
Alternative: Passing of parameters in registers ... :-)



The values of the actual parameters are determined before allocation of the new stack frame.



The complete frame is allocated inside the new block – plus the space for the current parameters.



Inside the new block, though, we must store the old SP (possibly +1) in order to correctly return the result ... :-)

3. Idea: Hybrid Solution

- For the first k threads, we allocate a separate stack area.
- For all further threads, we successively use one of the existing ones !!!

$$\Longrightarrow$$

- For few threads extremely simple and efficient;
- For many threads amortized storage usage :-))