# 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.

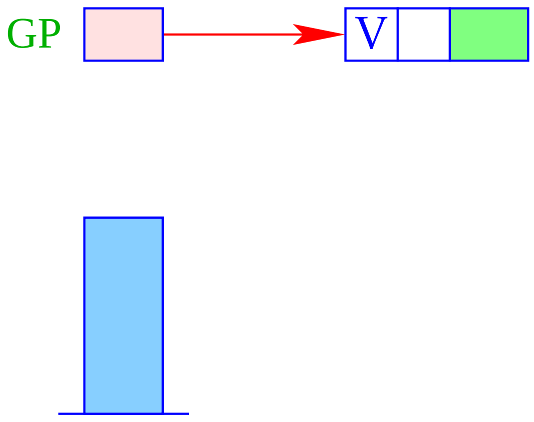
  Recall, e.g., the construction of a closure for an expression $e$ ...

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

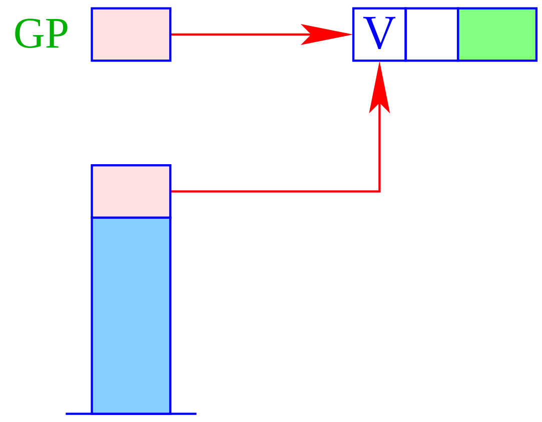
where $\{z_0, \ldots, z_{g-1}\} = \mathit{free}(e)$ and $\rho' = \{z_i \mapsto (G, i) \mid i = 0, \ldots, g-1\}$.

## 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 :




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:

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

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 $x$ $\rho$ sd | mkclos A | A: | pushglob 0 | | update |
| mkvec 1 | jump B | | eval | B: | ... |

## Example:

$e \equiv \textbf{letrec}\ a = b; b = 7\ \textbf{in}\ a.$ $\quad \text{code}_V\ e\ \emptyset\ 0$ 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 |

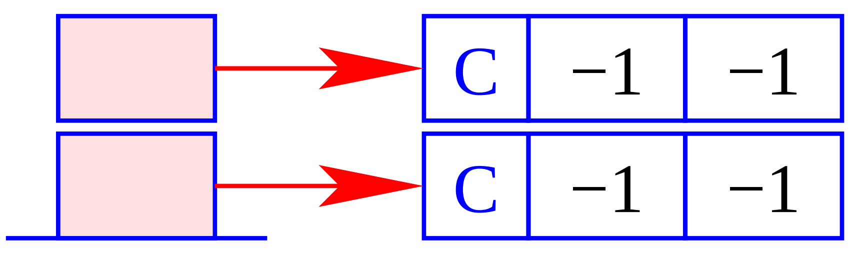
The execution of this instruction sequence should deliver the basic value 7 ...

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 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 |

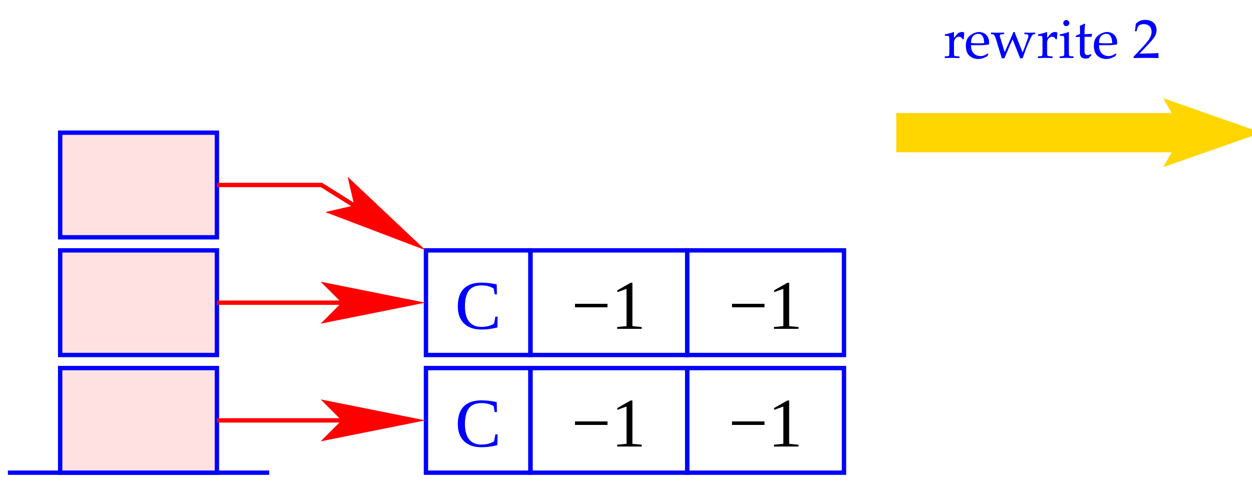
alloc 2

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 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 |

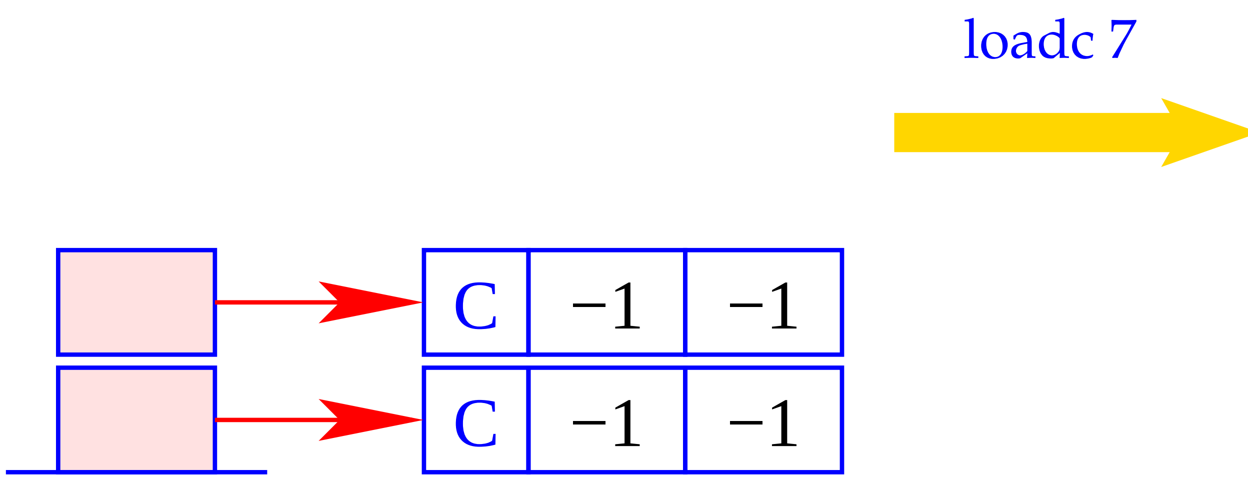
pushloc 0



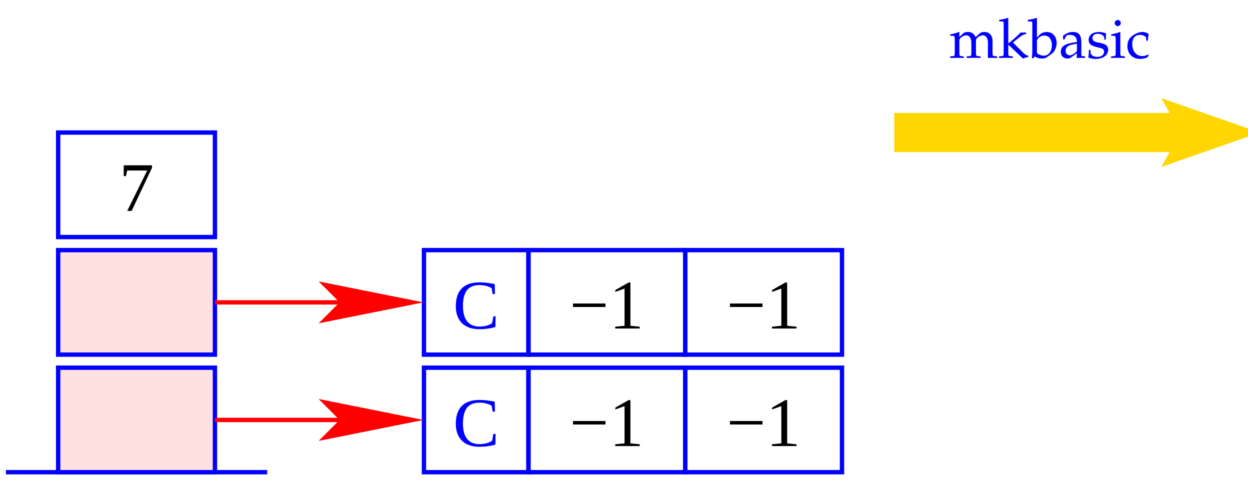
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | alloc 2 | 3 | rewrite 2 | 3 | mkbasic | 2 | pushloc 1 |
| 2 | pushloc 0 | 2 | loadc 7 | 3 | rewrite 1 | 3 | eval |
| | | | | | | 3 | slide 2 |



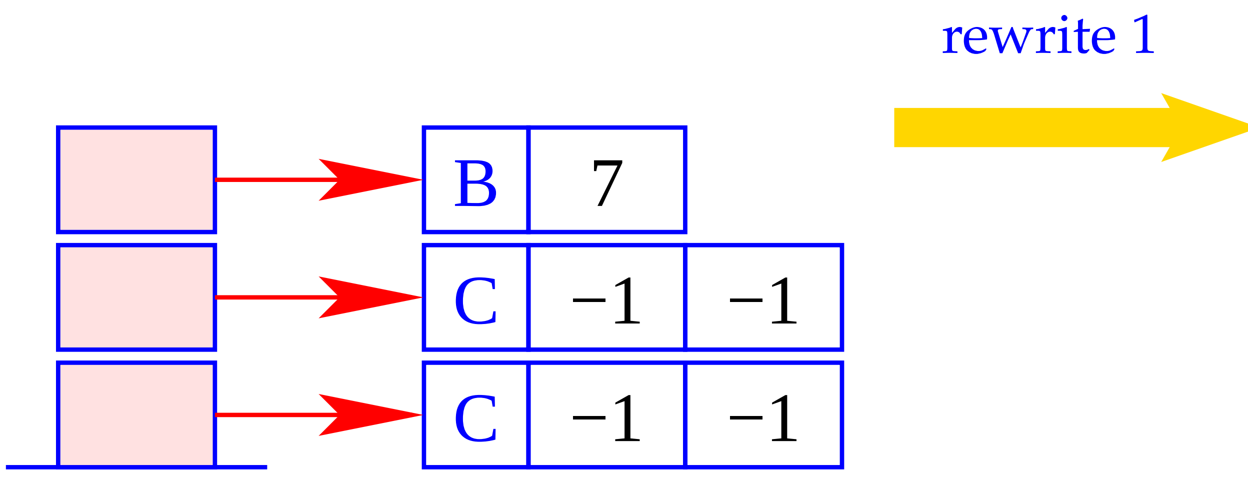
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | alloc 2 | 3 | rewrite 2 | 3 | mkbasic | 2 | pushloc 1 |
| 2 | pushloc 0 | 2 | loadc 7 | 3 | rewrite 1 | 3 | eval |
| | | | | | | 3 | slide 2 |



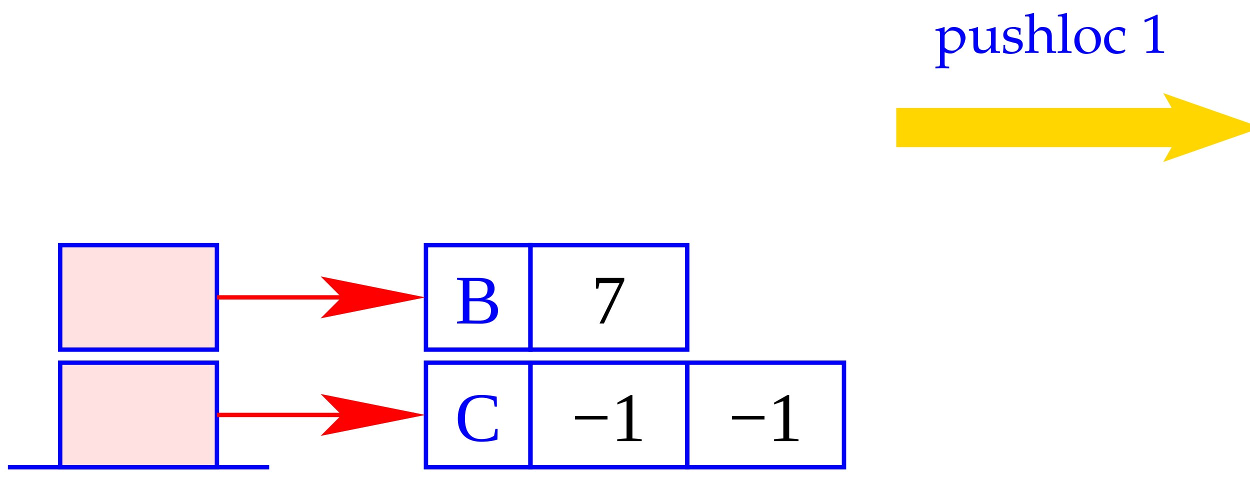
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | alloc 2 | 3 | rewrite 2 | 3 | mkbasic | 2 | pushloc 1 |
| 2 | pushloc 0 | 2 | loadc 7 | 3 | rewrite 1 | 3 | eval |
| | | | | | | 3 | slide 2 |



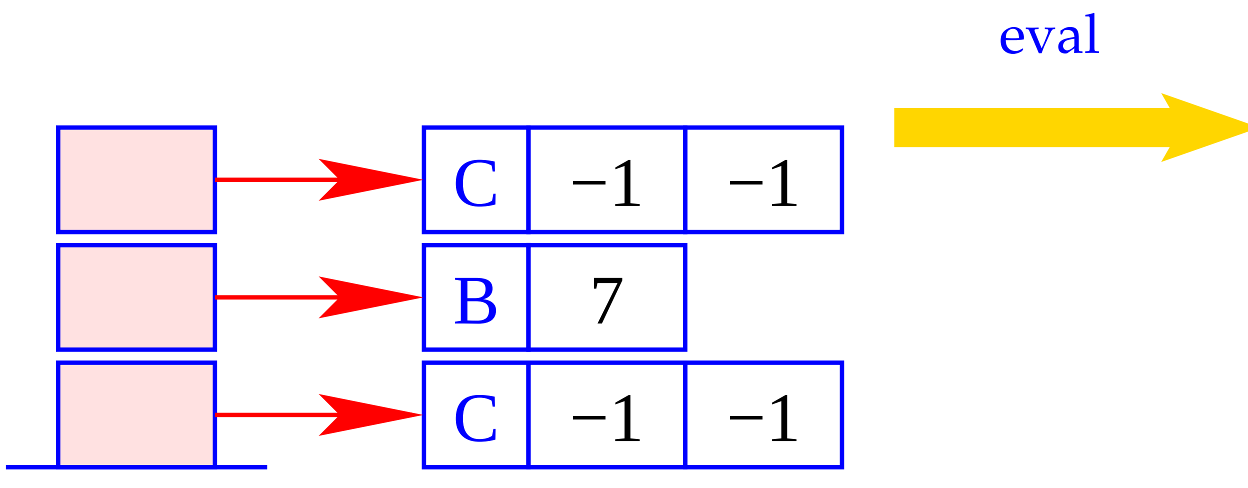
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | alloc 2 | 3 | rewrite 2 | 3 | mkbasic | 2 | pushloc 1 |
| 2 | pushloc 0 | 2 | loadc 7 | 3 | rewrite 1 | 3 | eval |
| | | | | | | 3 | slide 2 |



| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 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 |



| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 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 |

# 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

$\textbf{let } a = b; \ldots b = a \textbf{ in } \ldots.$

**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:

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

# 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.

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

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

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:

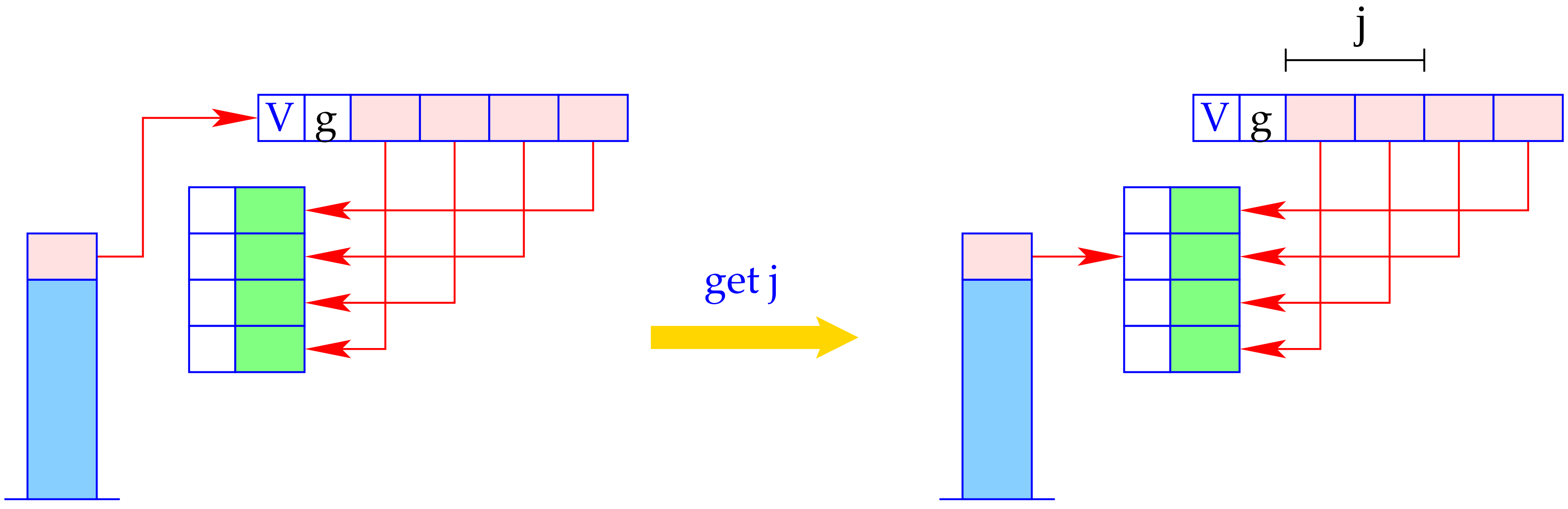
$$\begin{array}{lcl} e & ::= & \ldots \mid (e_0,\ldots,e_{k-1}) \mid \#j\; e \\ & & \mid\; \textbf{let}\;(x_0,\ldots,x_{k-1}) = e_1 \;\textbf{in}\; e_0 \end{array}$$

- 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:

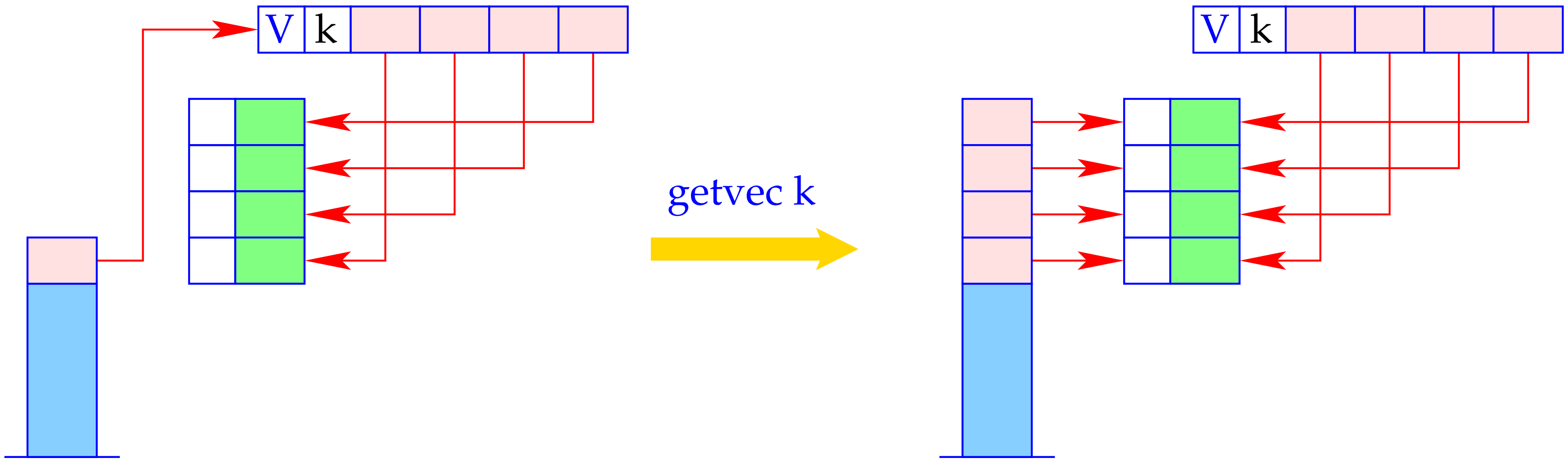
$$e \equiv \textbf{let}\ (y_0, \ldots, y_{k-1}) = e_1\ \textbf{in}\ e_0$$

This is translated as follows:

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

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 ...

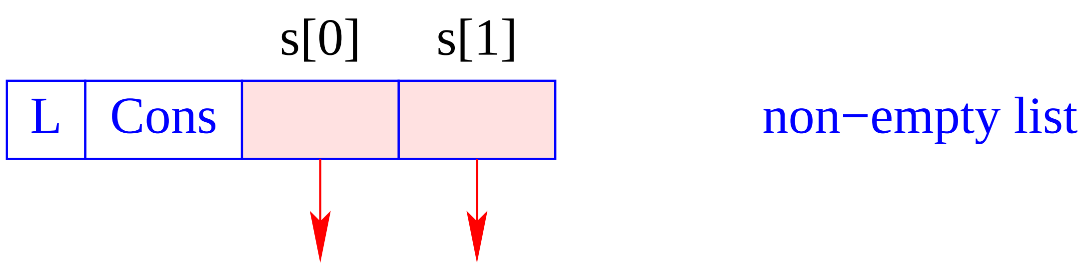
Example: The append function app:

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

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

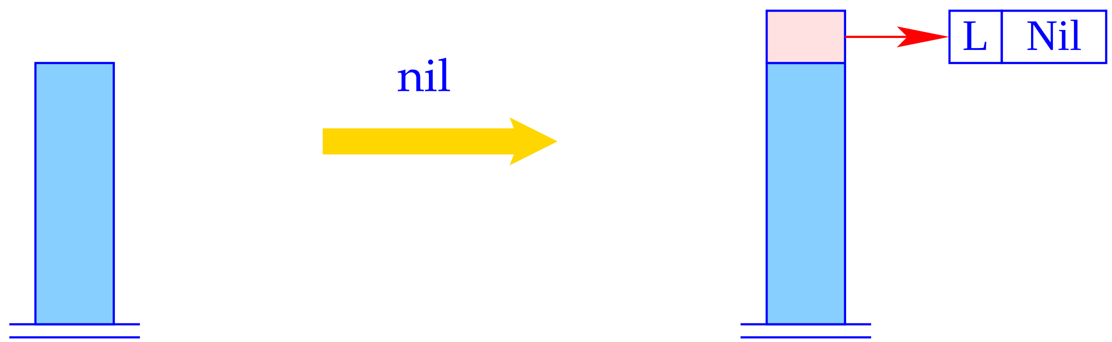
The new instructions nil and cons are introduced for building list nodes.

We translate for CBN:

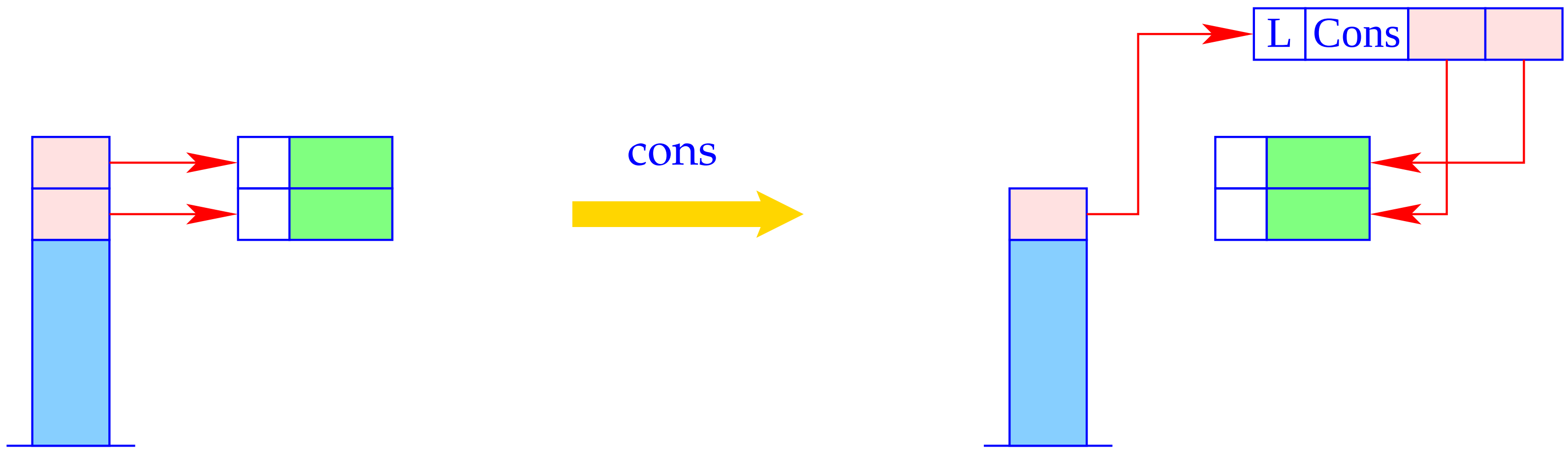
$$\begin{array}{lll} \text{code}_V\ []\ \rho\ \text{sd} & = & \text{nil} \\ \text{code}_V\ (e_1 : e_2)\ \rho\ \text{sd} & = & \text{code}_C\ e_1\ \rho\ \text{sd} \\ & & \text{code}_C\ e_2\ \rho\ (\text{sd}+1) \\ & & \text{cons} \end{array}$$

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:

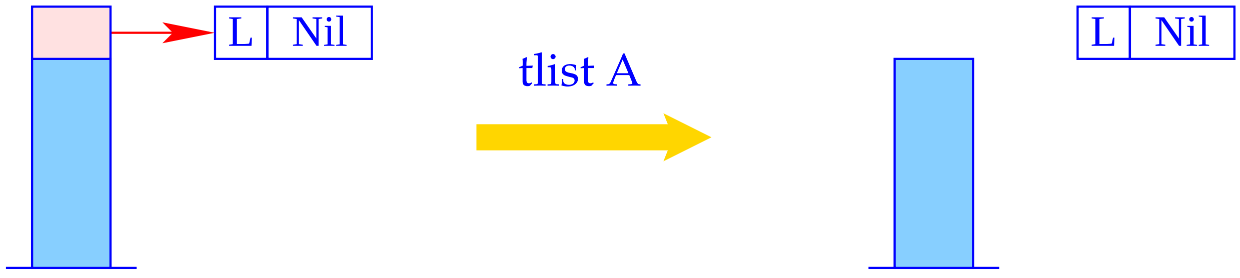
- evaluation of $e_0$;
- check, whether resulting value $v$ is an L-object;
- if $v$ is the empty list, evaluation of $e_1$ ...
- otherwise storing the two references of $v$ on the stack and evaluation of $e_2$. This corresponds to binding $h$ and $t$ to the two components of $v$.

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

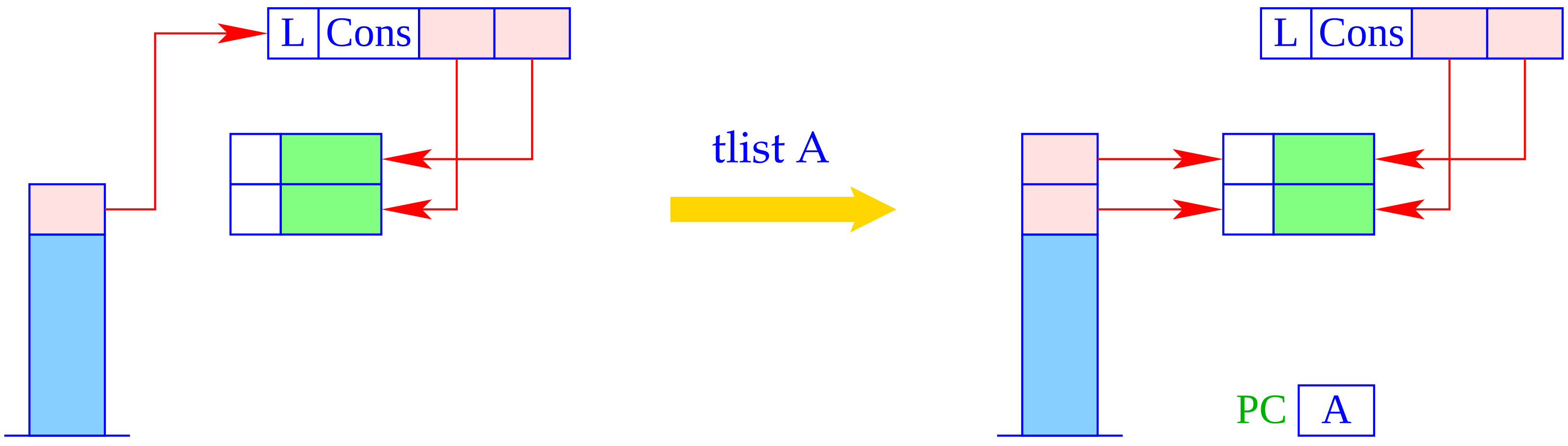
$$
\begin{array}{rcrl}
\text{code}_V\ e\ \rho\ \text{sd} & = & & \text{code}_V\ e_0\ \rho\ \text{sd} \\
& & & \text{tlist A} \\
& & & \text{code}_V\ e_1\ \rho\ \text{sd} \\
& & & \text{jump B} \\
& & \text{A}: & \text{code}_V\ e_2\ \rho'\ (\text{sd} + 2) \\
& & & \text{slide } 2 \\
& & \text{B}: & \ldots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

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 $\mathrm{app} \mapsto (G, 0)$ :

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 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 $\text{code}_C$ can be optimized for tuples and lists:

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

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

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

**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.

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

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

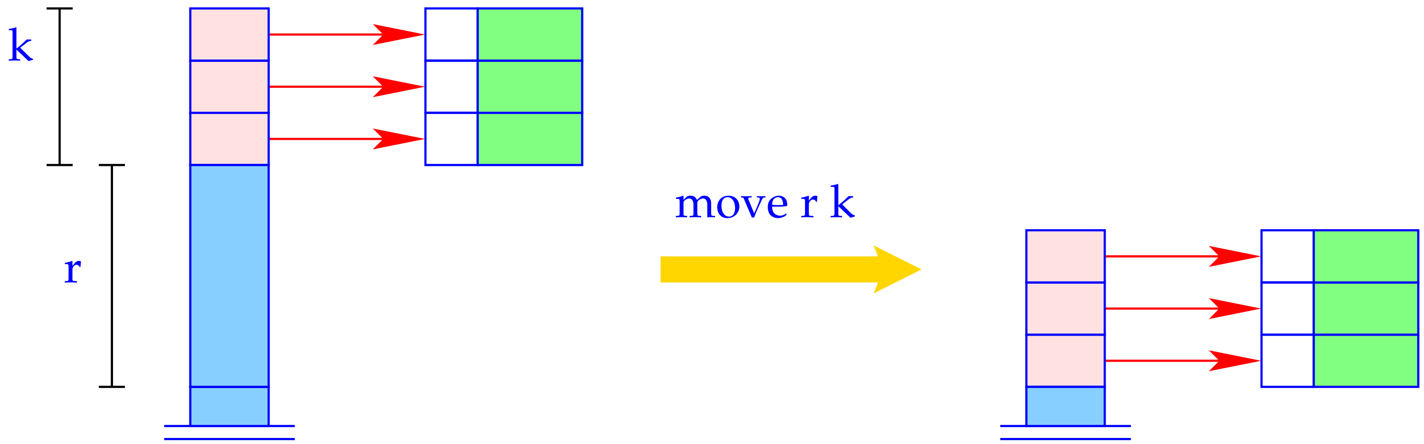
## Example:

The body of the function

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

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 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;