# 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:

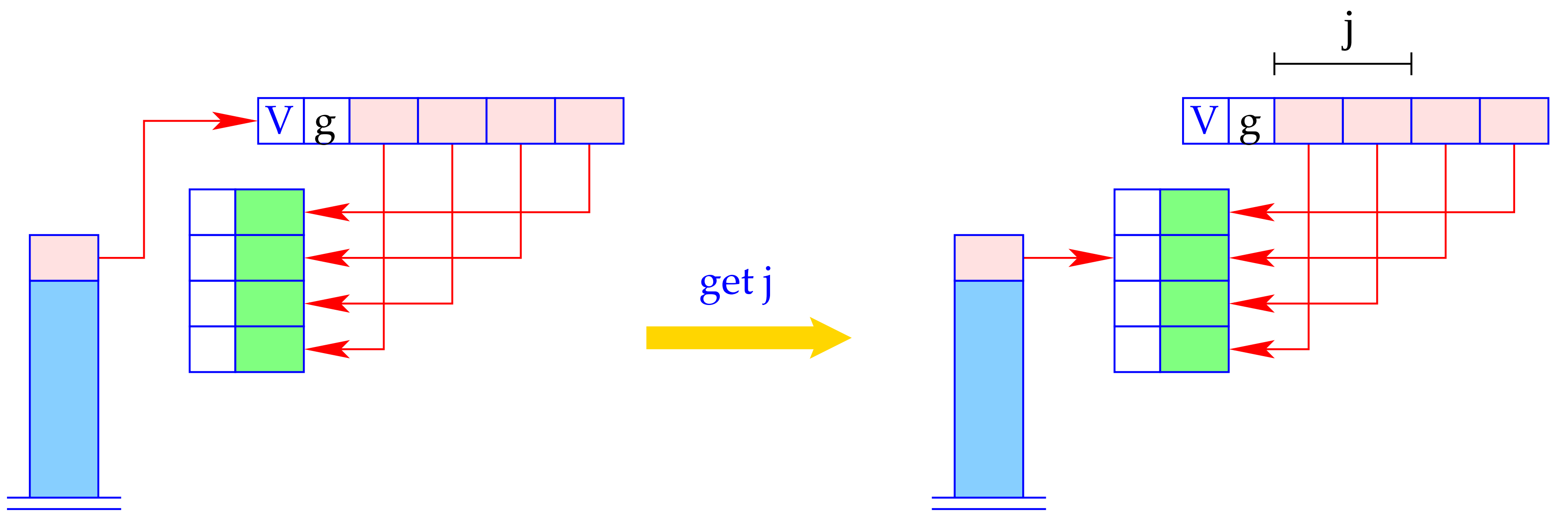
$$\begin{array}{lcl} e & ::= & \ldots \mid (e_0, \ldots, e_{k-1}) \mid \#j\ e \\ & & \mid \ \mathbf{let}\ (x_0, \ldots, x_{k-1}) = e_1\ \mathbf{in}\ e_0 \end{array}$$

- 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.

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

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

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:

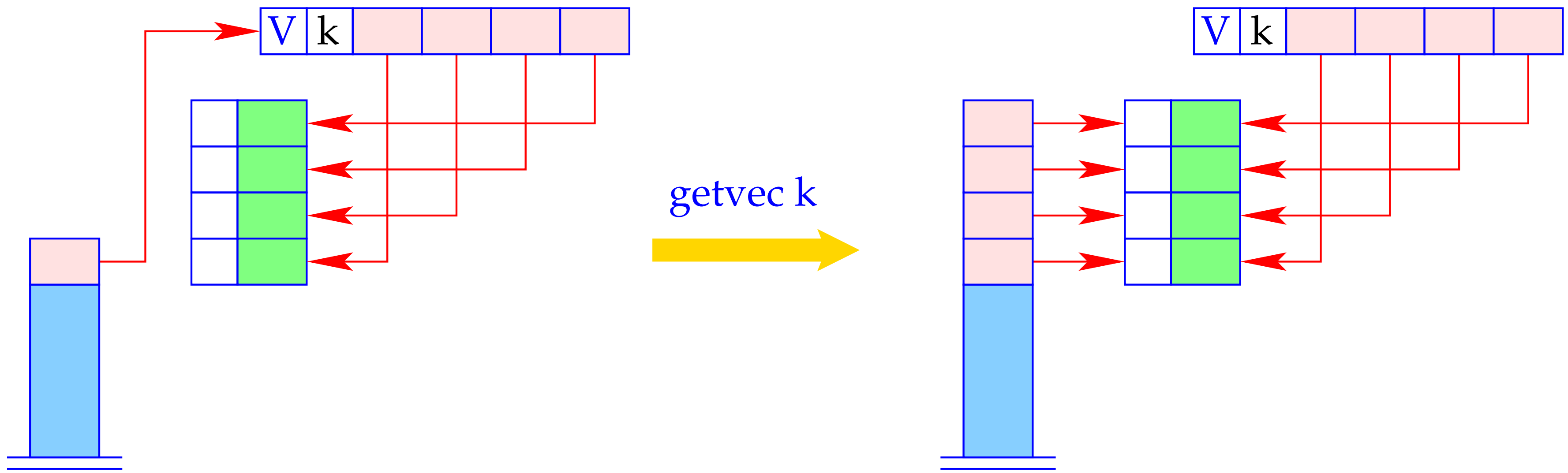
$$e \equiv \textbf{let } (y_0, \ldots, y_{k-1}) = e_1 \textbf{ in } e_0$$

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 ...

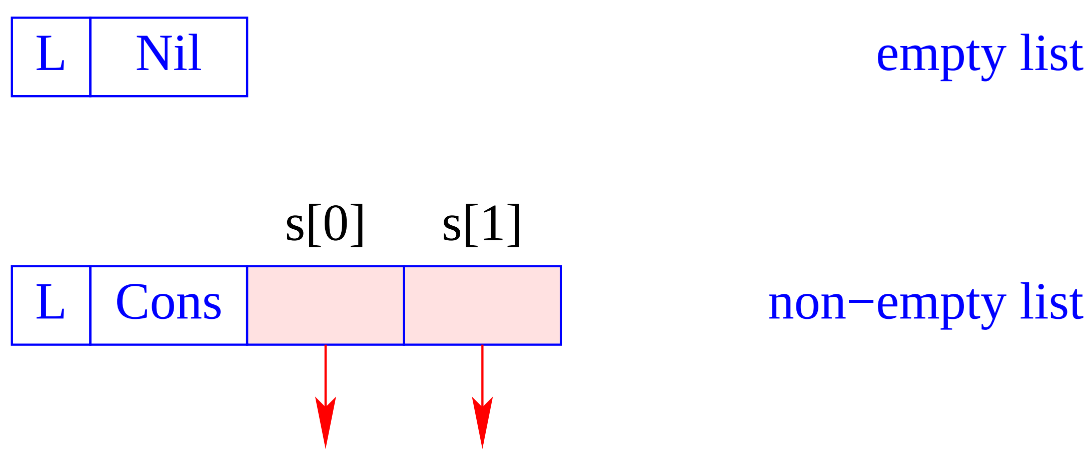
Example: The append function app:

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

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

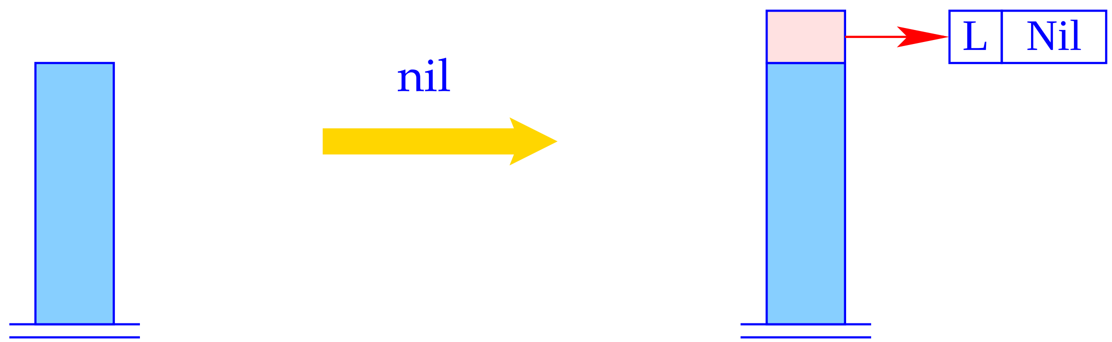
The new instructions nil and cons are introduced for building list nodes.

We translate for CBN:

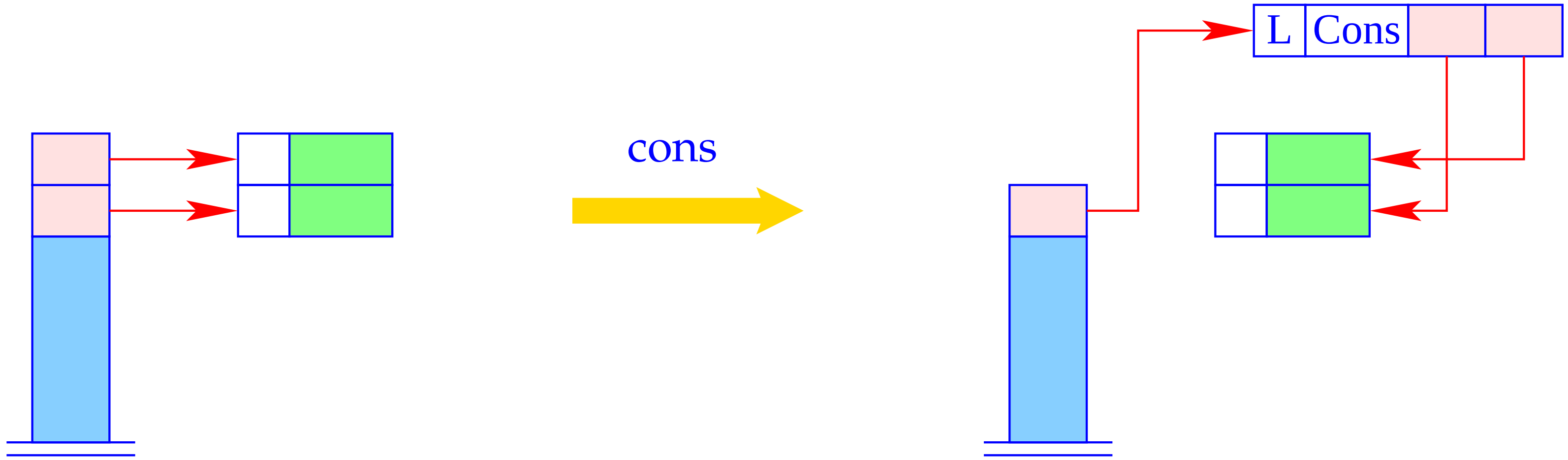
$$
\begin{array}{lll}
\text{code}_V\ []\ \rho\ \text{sd} & = & \text{nil} \\
\text{code}_V\ (e_1 : e_2)\ \rho\ \text{sd} & = & \text{code}_C\ e_1\ \rho\ \text{sd} \\
& & \text{code}_C\ e_2\ \rho\ (\text{sd}+1) \\
& & \text{cons}
\end{array}
$$

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:

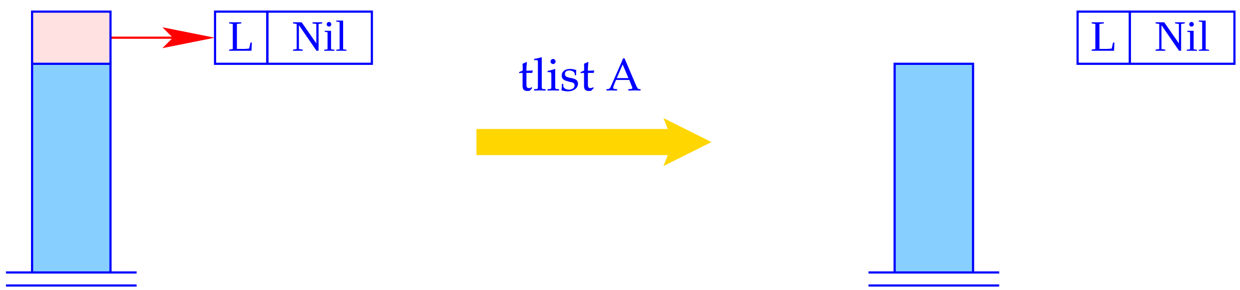
- evaluation of $e_0$;
- check, whether resulting value $v$ is an L-object;
- if $v$ is the empty list, evaluation of $e_1$ ...
- otherwise storing the two references of $v$ on the stack and evaluation of $e_2$. This corresponds to binding $h$ and $t$ to the two components of $v$.

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

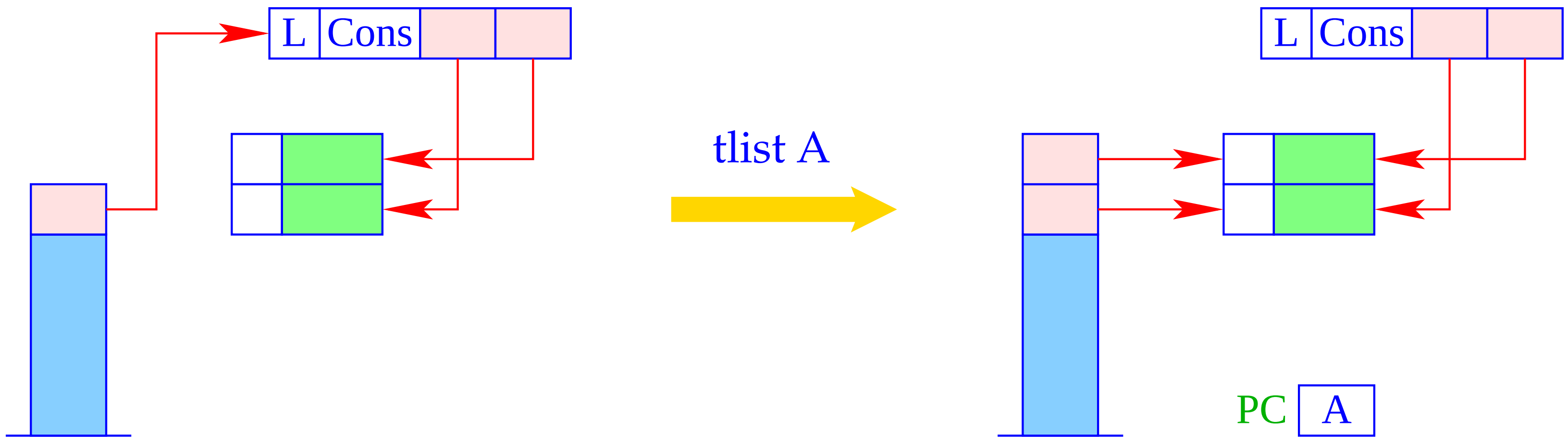
$$
\begin{array}{rrl}
\mathrm{code}_V\; e\; \rho\; \mathrm{sd} & = & \mathrm{code}_V\; e_0\; \rho\; \mathrm{sd} \\
& & \mathrm{tlist}\ \mathrm{A} \\
& & \mathrm{code}_V\; e_1\; \rho\; \mathrm{sd} \\
& & \mathrm{jump}\ \mathrm{B} \\
& \mathrm{A}: & \mathrm{code}_V\; e_2\; \rho'\; (\mathrm{sd}+2) \\
&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

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

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 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:

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

# 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 | $\textbf{case}\ x\ \textbf{of}\ [\,]\ \to y;\ \ h : t\ \to r\ t\ (h : y)$ |
| $f\ (x - 1)$ is not a last call in | $\textbf{if}\ x \leq 1\ \textbf{then}\ 1\ \textbf{else}\ x * f\ (x - 1)$ |

**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.

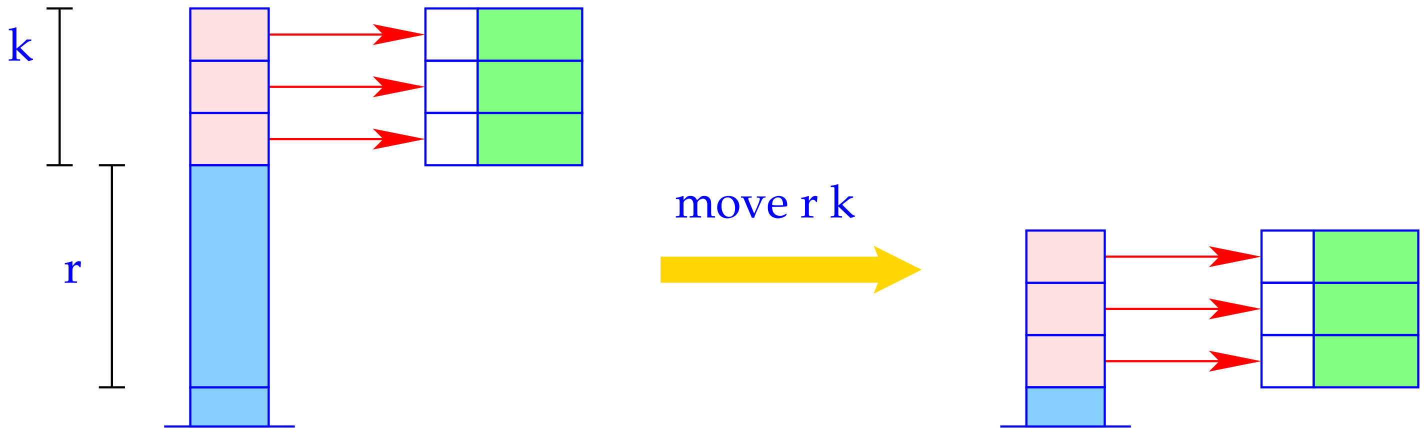
## Example:

The body of the function

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

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