MaMa — a simple abstract machine for functional languages

Functional Language PuF

We will consider a mini-language of "Pure Functions" PuF.

Programs are expressions e in form:

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

- For simplicity, the only primitive type is `int`.
- Later, we will add data structures.

Functional Language PuF

Example: factorial function:

$$\begin{array}{lcl} \mathsf{fac} & = & \mathbf{fn}\ x \Rightarrow \mathbf{if}\ x \leq 1\ \mathbf{then}\ 1 \\ & & \mathbf{else}\ x \cdot \mathsf{fac}\,(x - 1) \end{array}$$

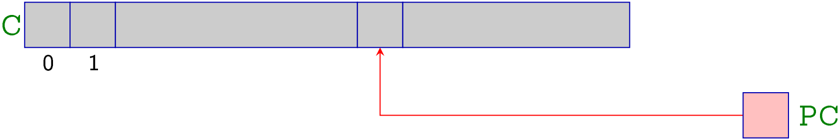
Functional languages use two different kinds of semantics:

CBV: call by value, arguments are evaluated before the evaluation of function body (eg. SML);

CBN: call by need, arguments are passed to the function as closures and are evaluated when their values are requested (eg. Haskell).

MaMa architecture

Code:

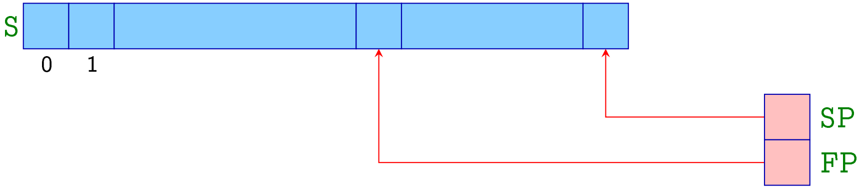


C	=	Code-store — memory area for a program code; each cell contains a single AM instruction.
PC	=	Program Counter — register containing an address of the instruction to be executed *next*.

Initially, PC contains the address 0; ie. C[0] contains the first instruction of the program.

MaMa architecture

Stack:



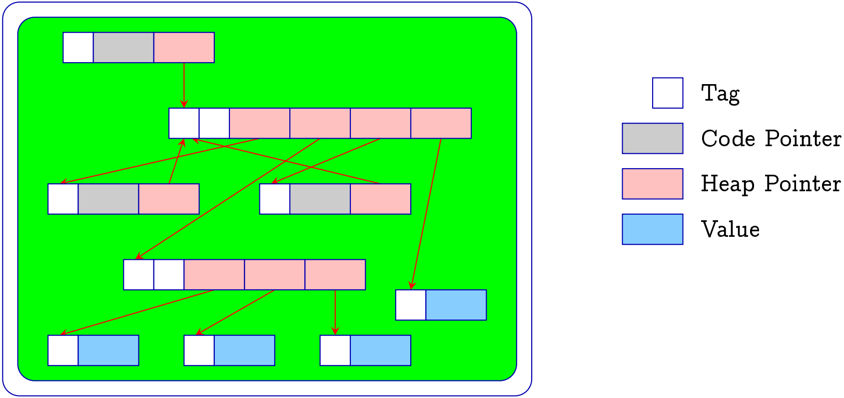
S = Stack — each cell contains a primitive value or an address;

SP = Stack-Pointer — points to top of the stack;

FP = Frame-Pointer — points to the currently active frame.

MaMa architecture

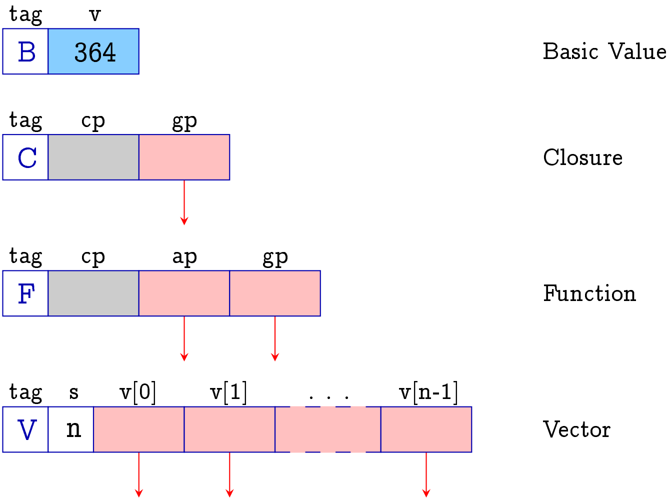
Heap:



H = Heap — memory area for dynamically allocated data.

MaMa architecture

Heap may contain following objects:



tag v
B 364
Basic Value

tag cp gp
C
Closure

tag cp ap gp
F
Function

tag s v[0] v[1] . . . v[n-1]
V n
Vector

MaMa architecture

Instruction **new**(tag, $args$) creates an object of the given kind and returns a pointer to it.

We will use the following three functions for code generation:

- $\text{code}_B\ e$ — evaluates an expression e of primitive type and saves its value into top of the stack;
- $\text{code}_V\ e$ — evaluates an expression e, saves it into the heap, and puts a pointer of it into top of the stack;
- $\text{code}_C\ e$ — does not evaluate an expression, but creates a closure of e in the heap and returns the pointer to it to top of the stack.

Simple expressions

Expression which are constructed only using constants, operator applications and conditional expressions are compiled analogously imperative languages:

$$
\begin{array}{lcl}
\mathrm{code}_B\ b\ \rho\ \mathrm{sd} & = & \texttt{loadc b} \\
\mathrm{code}_B\ (\Box_1\ e)\ \rho\ \mathrm{sd} & = & \mathrm{code}_B\ e\ \rho\ \mathrm{sd} \\
 & & \mathrm{op}_1 \\
\mathrm{code}_B\ (e_1\,\Box_2\,e_2)\ \rho\ \mathrm{sd} & = & \mathrm{code}_B\ e_1\ \rho\ \mathrm{sd} \\
 & & \mathrm{code}_B\ e_2\ \rho\ (\mathrm{sd}+1) \\
 & & \mathrm{op}_2
\end{array}
$$

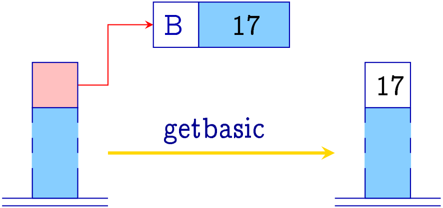
Simple expressions

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

In the case of other forms of expressions, we first compute its value in the heap and load the value by dereferencing the returned pointer:

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

Simple expressions



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

- ρ denotes an *address environment* which is in the form:

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

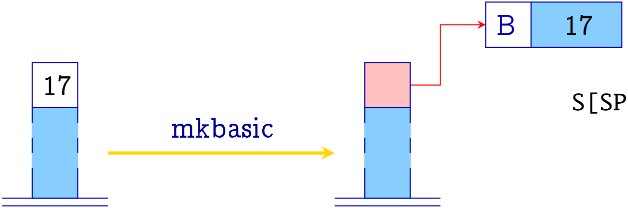
- An extra parameter sd (stack difference) simulates the change of register SP by instructions which modify the stack. We'll use it later for variable addressing.

Simple expressions

Function $\mathtt{code}_V$ for simple expressions is analogous to $\mathtt{code}_B$ but creates an object for the primitive value in the heap.

$$
\begin{array}{lcl}
\mathtt{code}_V\ b\ \rho\ \mathrm{sd} & = & \texttt{loadc b} \\
 & & \texttt{mkbasic} \\
\mathtt{code}_V\ (\Box_1\, e)\ \rho\ \mathrm{sd} & = & \mathtt{code}_B\ e\ \rho\ \mathrm{sd} \\
 & & \texttt{op}_1 \\
 & & \texttt{mkbasic} \\
\mathtt{code}_V\ (e_1\, \Box_2\, e_2)\ \rho\ \mathrm{sd} & = & \mathtt{code}_B\ e_1\ \rho\ \mathrm{sd} \\
 & & \mathtt{code}_B\ e_2\ \rho\ (\mathrm{sd}+1) \\
 & & \texttt{op}_2 \\
 & & \texttt{mkbasic}
\end{array}
$$

Simple expressions



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

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

Variables

Example: consider definitions

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

Function f uses a *global* variable c and *local* variables a (formal parameter) and b (defined by the inner **let**-expression).

A value of the global variable is determined during the *construction* of the function (static scoping!) and is directly accessible during execution.

Variables

Global variables

- Values corresponding to global variables are kept in the heap as a vector (Global Vector).
- They are addressed sequentially starting from 0.
- During the construction of F-object or C-object, its global vector is built and the address to it is put into objects gp-field.
- During evaluation, the register GP (Global Pointer) points to the currently active global vector.

Variables

Local variables

Local variables are kept in a stack frame.

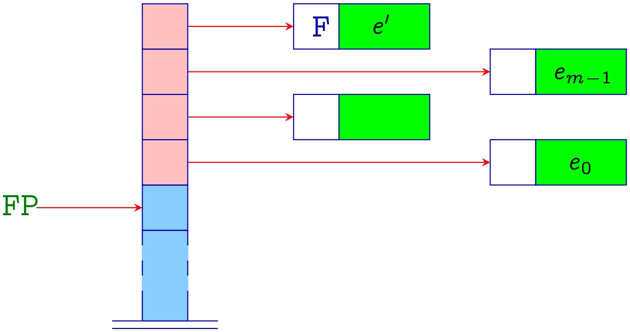
Let $e \equiv e' \; e_0 \; \ldots \; e_{m-1}$ be an application of the function e' to arguments $e_0, \ldots, e_{m-1}$.

NB! The arity of function e' can be different of m.

- PuF functions are curried $f : t_1 \to t_2 \to \cdots \to t_n \to t$.
- Hence f may be applied to less than n arguments (*partial application*).
- If t is a function type, then f may be applied to more than n arguments.

Variables

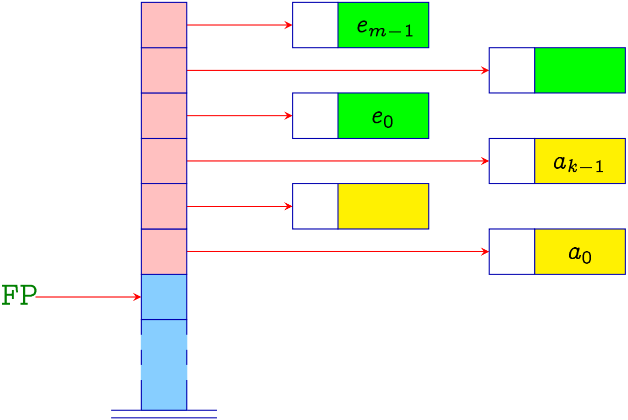
Possible organization of a stack frame:



- \+ Parameters can be addressed relative to FP.
- − Local variables of e' can't be addressed relative to FP.
- − If e' is n-ary function and $n < m$, then the rest of $m - n$ arguments must be relocated in the frame.

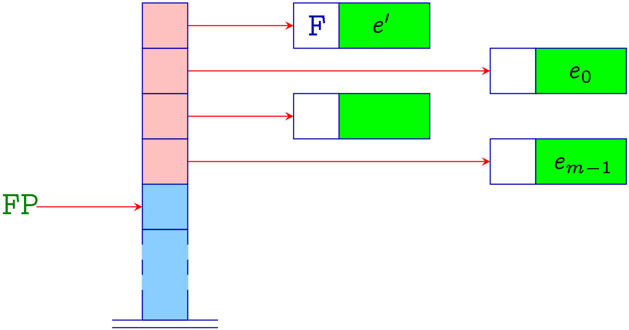
Variables

- If e' evaluates to a function which is already partially applied to arguments $a_0, \ldots, a_{k-1}$ then these arguments must be moved downwards under e_0.



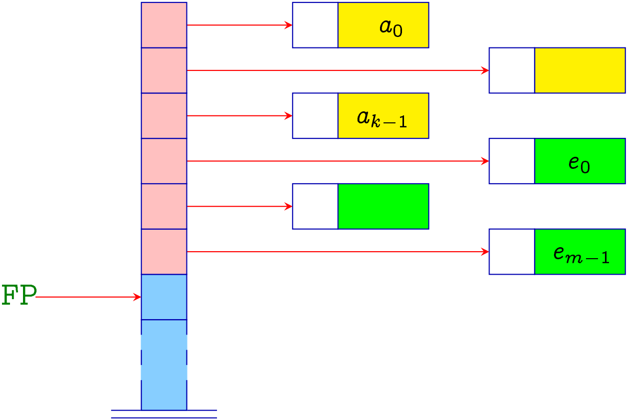
Variables

Alternative organization of a stack frame:



- Additional parameters $a_0, \ldots, a_{k-1}$ and local variables can be pushed to the stack after arguments.

Variables

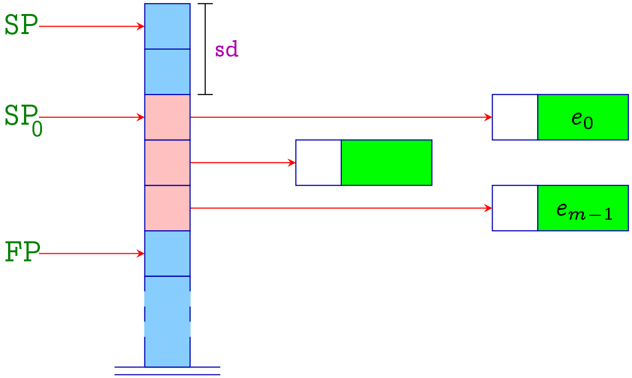


- Addressing of formal parameters relative of FP is not possible anymore.

Variables

Solution:

- Addressing both arguments and local variables relative of SP!
- However SP is changing during the execution ...



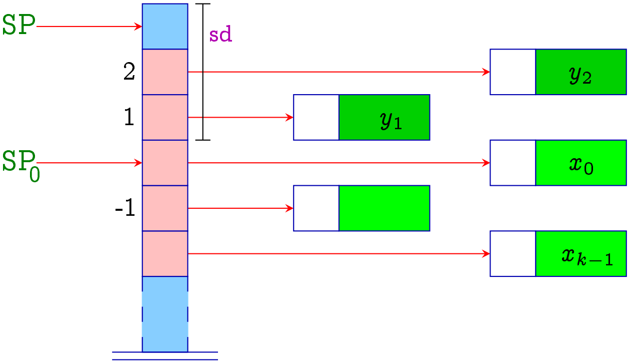
Variables

- Stack difference, sd, describes the difference of the current value of SP from its value SP_0 at the entering into the function.
- The difference can be determined statically by simulating stack modifications by instructions.
- Formal parameters $x_0, x_1, x_2, \ldots$ are bound to non positive relative addresses 0, -1, -2, . . . ; ie. $\rho\ x_i = (L, -i)$.
- The absolute address of i-th formal parameter is:

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

- Local **let**-variables are pushed sequentially to top of the stack.

Variables



- Local variables $y_1, y_2, \ldots$ are bound to positive relative addresses; ie. $\rho\ y_i = (L, i)$.
- The absolute address of i-th local variable is:

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

Variables

The evaluation of variables in CBN semantics:

$$\begin{array}{llll} \mathtt{code}_V\ x\ \rho\ \mathsf{sd} & = & \mathtt{pushloc}\ (\mathsf{sd} - i) & \text{if } \rho\ x = (L, i) \\ & & \mathtt{eval} & \\ \mathtt{code}_V\ x\ \rho\ \mathsf{sd} & = & \mathtt{pushglob\ i} & \text{if } \rho\ x = (G, i) \\ & & \mathtt{eval} & \end{array}$$

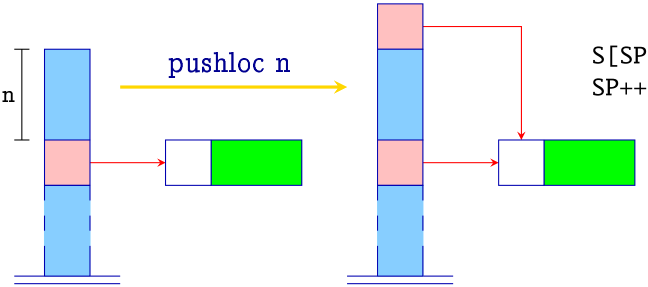
Instruction `eval` checks whether the variable is already evaluated or not, and if not, forces its evaluation (will be considered later).

In case of CBV semantics there is no need for `eval` instruction.

Variables

A local variable with a relative address i corresponds to the stack cell $S[a]$, where

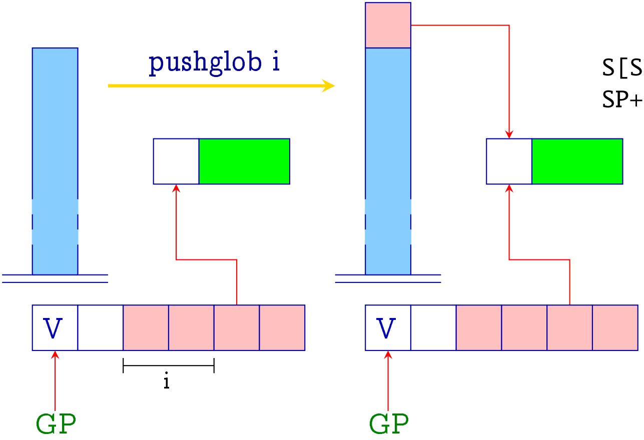
$$a = \mathrm{SP} - (\mathrm{sd} - i) = (\mathrm{SP} - \mathrm{sd}) + i = \mathrm{SP}_0 + i$$



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

Variables

Global variables are in the global vector.



```
S[SP+1] = GP→v[i];
SP++;
```

Variables

Example:

Let $e \equiv (b + c)$ with environment $\rho = \{b \mapsto (L, 1), c \mapsto (G, 0)\}$ and $\text{sd} = 1$.

In case of CBN semantics $\text{code}_V\ e\ \rho\ \text{sd}$ emits the code:

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

Function definitions

Compilation of a function definition generates a code which constructs a *functional value* in the heap:

- creates a global vector for global variables;
- creates an (initially empty) argument vector;
- creates a F-object, which contains pointers to these vectors and a pointer to the start address of the code corresponding to function body.

The code for function body is generated separately.

Function definitions

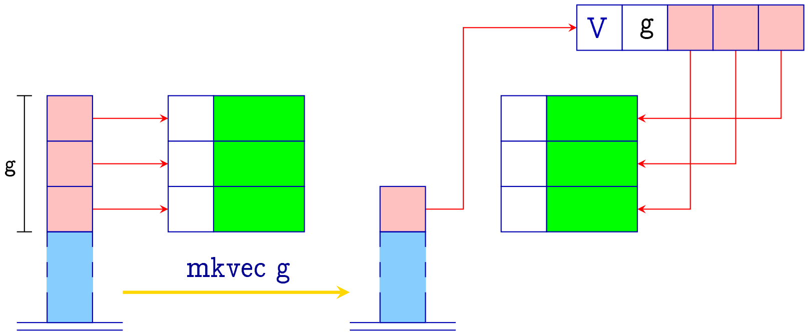
$\mathrm{code}_V\ (\mathbf{fn}\ x_0, \ldots, x_{k-1} \Rightarrow e)\ \rho\ \mathrm{sd} \quad =$

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

$$
\begin{aligned}
\text{where}\quad \{z_0, \ldots, z_{g-1}\} &= \mathit{free}(\mathbf{fn}\ x_0, \ldots, x_{k-1} \Rightarrow e)\\
\rho' &= \{x_i \mapsto (L, -i) \mid i = 0, \ldots, k-1\}\\
&\cup\ \{z_j \mapsto (G, j) \mid j = 0, \ldots, g-1\}
\end{aligned}
$$

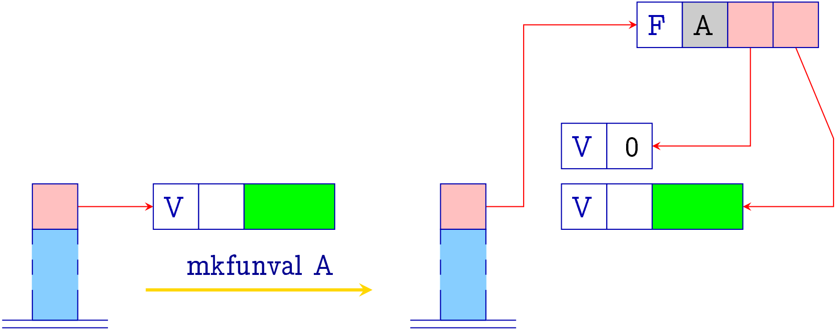
$$
\mathrm{getvar}\ y\ \rho\ \mathrm{sd} = \begin{cases} \texttt{pushloc}\ (\mathrm{sd} - i) & \text{if } \rho\ y = (L, i)\\ \texttt{pushglob}\ \texttt{j} & \text{if } \rho\ y = (G, j) \end{cases}
$$

Function definitions



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

Function definitions



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

Function definitions

Example: let $f \equiv \mathbf{fn}\ b \Rightarrow a + b$ with environment $\rho = \{a \mapsto (L, 1)\}$ and $\mathsf{sd} = 2$.

$\mathsf{code}_V\ f\ \rho\ \mathsf{sd}$ emits a code:

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

Instructions `targ k` and `return k` are considered later.

Function applications

For function application $e' \; e_0 \; \ldots \; e_{m-1}$ code is generated, which:

- creates a new frame in the stack;
- passes actual parameters; ie.

  CBV: evaluates actual parameters;
  CBN: creates closures of actual parameters;
- evaluates the function e' into F-object;
- applies the function to its arguments.

Function applications

In case of CBN semantics the following code is generated:

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

CBV uses $\texttt{code}_V$ instead of $\texttt{code}_C$ for arguments e_i.

Function applications

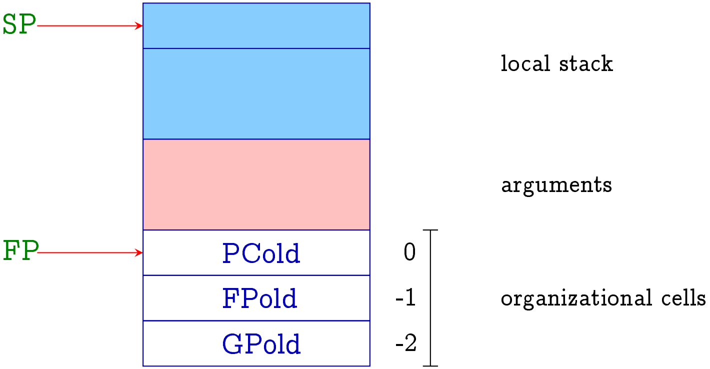
Example: let $e \equiv f\ 42$ with environment $\rho = \{f \mapsto (L, 2)\}$ and sd $= 2$.

$\text{code}_V\ e\ \rho$ sd emits a code (for CBV):

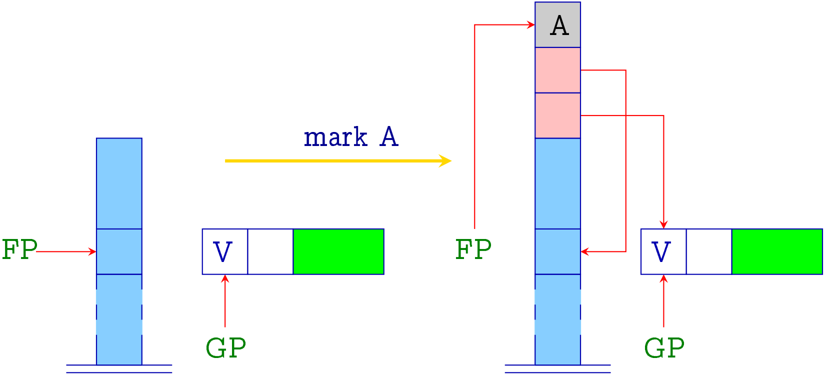
```
2     mark A              6        pushloc 4
5     loadc 42            7        apply
6     mkbasic             3     A: ...
```

Function applications

Structure of a frame:

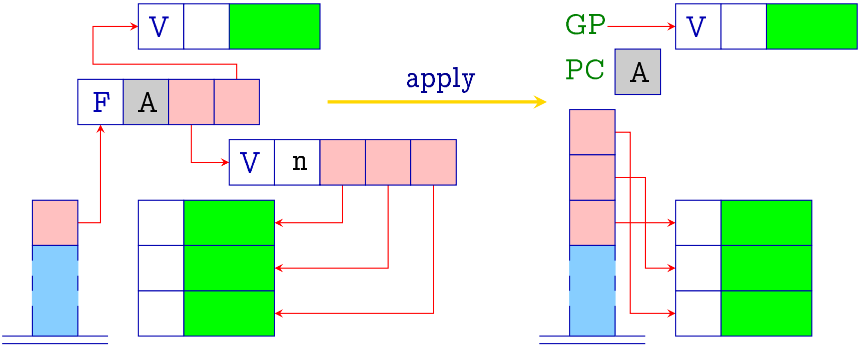


Function applications



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

Function applications



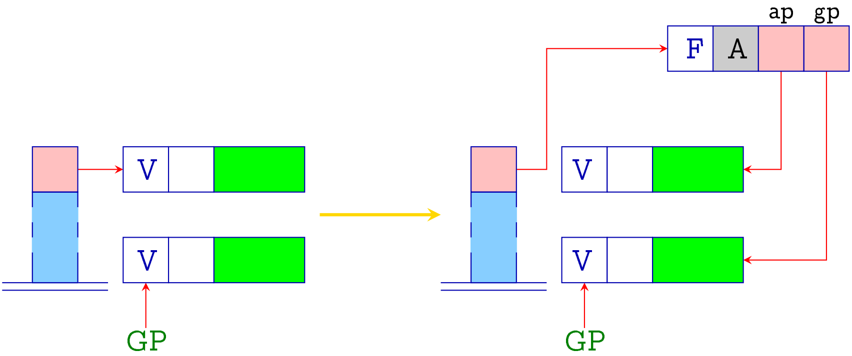
```
h = S[SP];                              GP = h→gp; PC = h→cp;
if (h→tag ≠ F)                          for (i=0; i < h→ap→n; i++)
  Error("Not Function");                  S[SP+i] = h→ap→v[i];
else {                                  SP = SP + h→ap→n - 1;
                                      }
```

Under- and oversupply of arguments

- The first instruction after `apply` is `targ k`.
- Checks whether there are enough arguments for the function application
  - uses the condition $\mathrm{SP} - \mathrm{FP} \geq k$.
- If there are enough arguments, starts the execution of the function body.
- Otherwise, creates a new functional value:
  - creates an argument vector;
  - creates a new F-object;
  - deallocates a frame in the stack.

Under- and oversupply of arguments

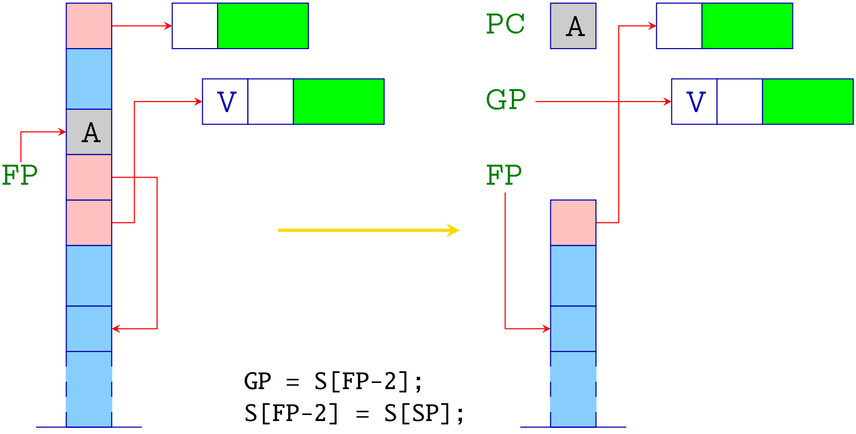
Construction of F-object:



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

Under- and oversupply of arguments

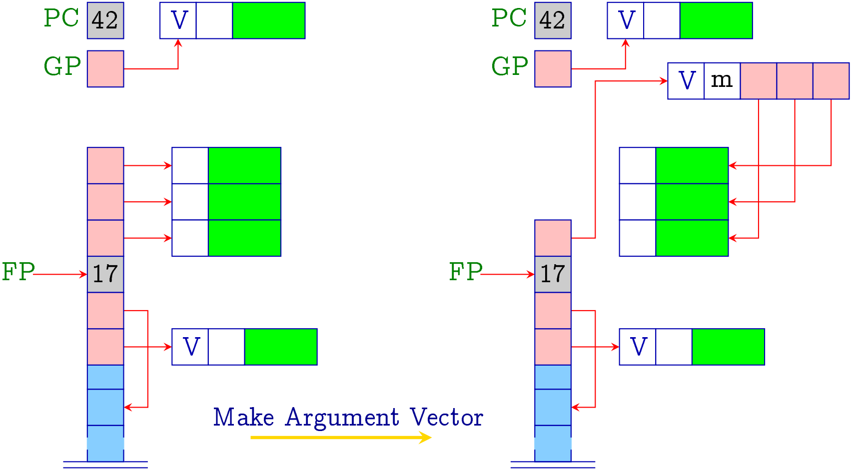
Releasing a stack frame:



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

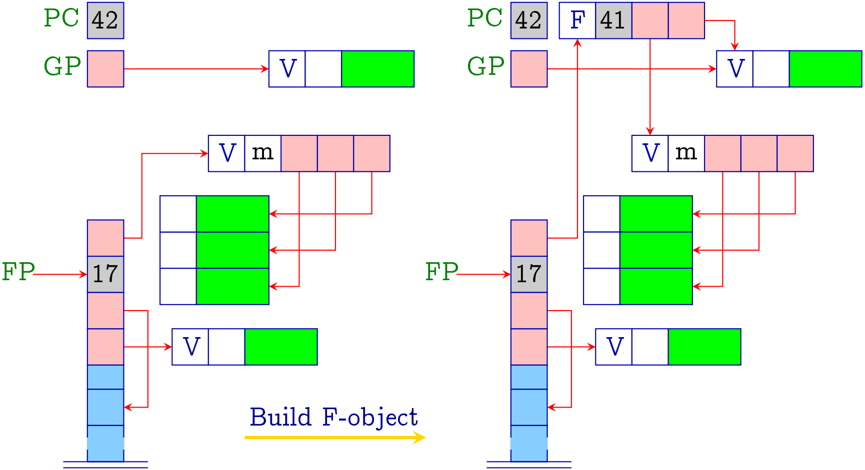
Under- and oversupply of arguments

`targ k`, if there are $m < k$ arguments



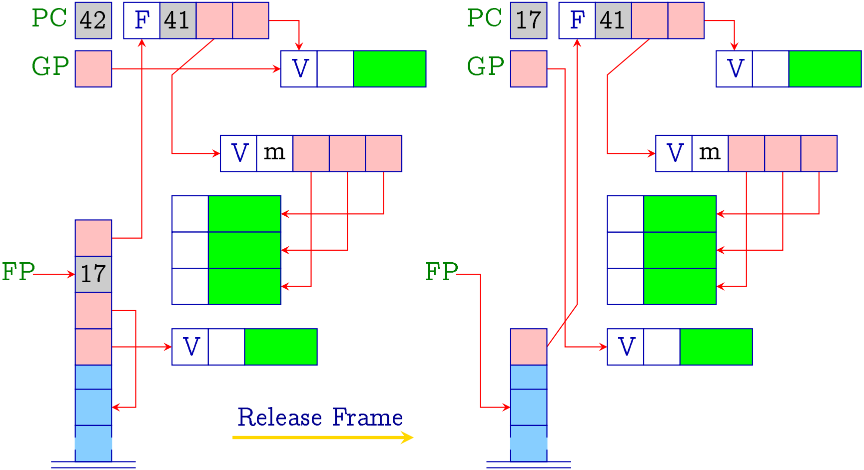
Under- and oversupply of arguments

`targ k`, if there are $m < k$ arguments



Under- and oversupply of arguments

`targ k`, if there are $m < k$ arguments



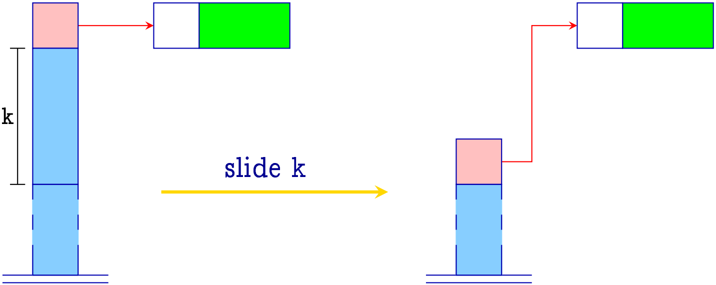
Under- and oversupply of arguments

- The last instruction of the function body, `return k`, checks whether the number of arguments is correct.
- If it is the case, then the frame is freed.
- Otherwise, the function had to evaluate into a new function which consumes the remaining arguments.

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

Under- and oversupply of arguments

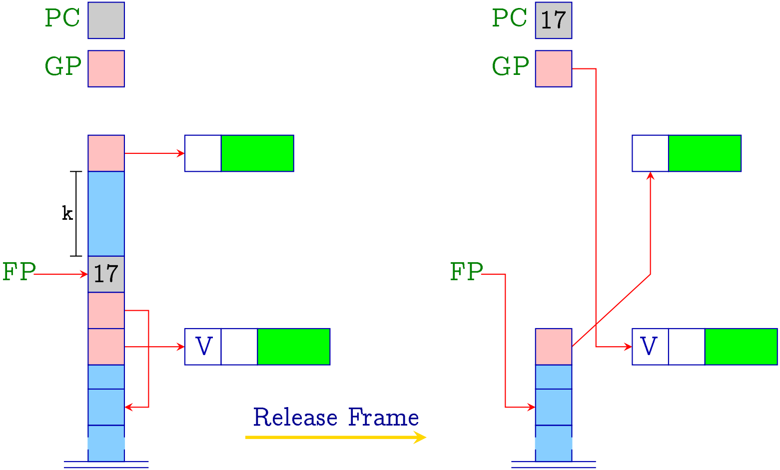
Instruction `slide k` moves top of the stack k cells downwards removing cells in between:



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

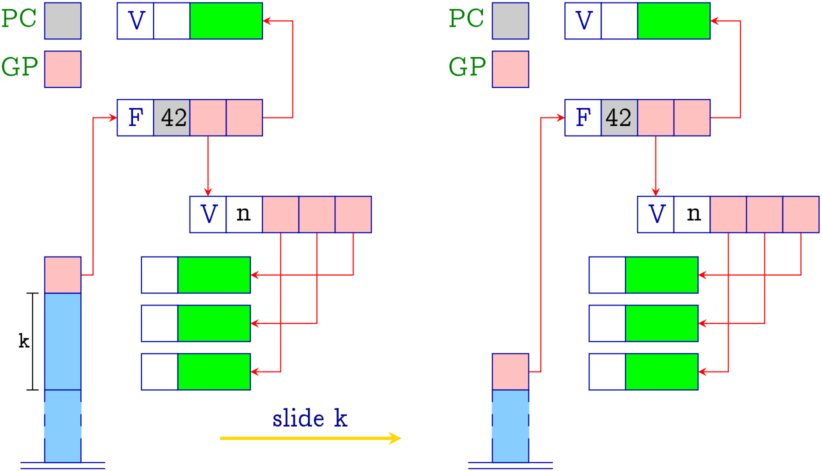
Under- and oversupply of arguments

`return k`, if there are *k* arguments



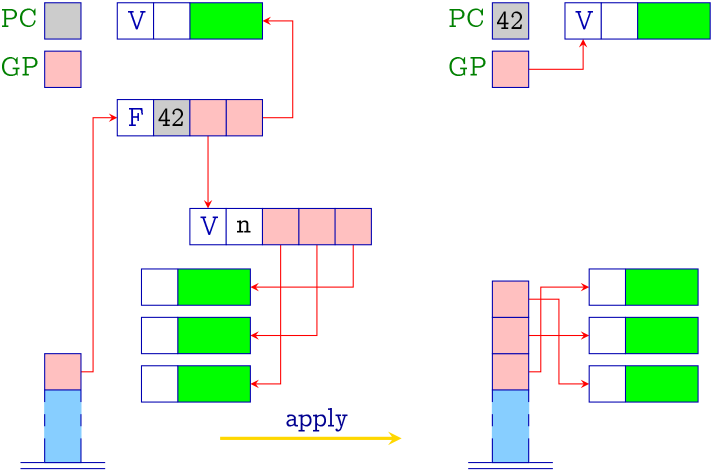
Under- and oversupply of arguments

`return k`, if there are $m > k$ arguments



Under- and oversupply of arguments

`return k`, if there are $m > k$ arguments



Local definitions

In case of a let-expression **let** $y_1 = e_1;\ \ldots;\ y_n = e_n$ **in** e_0, the generated code:

- binds variables $y_1, \ldots, y_n$ with corresponding values; ie.
  - CBV: evaluates expressions $e_1, \ldots, e_n$ and binds variables with their values;
  - CBN: binds variables with closures of expressions $e_1, \ldots, e_n$;
- evaluates an expression e_0 and returns its value.

In letrec-expression **letrec** $y_1 = e_1;\ \ldots;\ y_n = e_n$ **in** e_0 expressions e_i may refer to variables y_j before their creation:

- variables are bound first to fictional values, which are later changed to actual ones.

Local definitions

In case of CBN semantics the following code is generated:

$$
\begin{array}{l}
\mathtt{code}_V\;(\mathbf{let}\;y_1 = e_1;\;\ldots;\;y_n = e_n\;\mathbf{in}\;e_0)\;\rho\;\mathtt{sd} \quad = \\
\qquad\qquad \mathtt{code}_C\;e_1\;\rho\;\mathtt{sd} \\
\qquad\qquad \mathtt{code}_C\;e_2\;\rho_1\;(\mathtt{sd}+1) \\
\qquad\qquad \ldots \\
\qquad\qquad \mathtt{code}_C\;e_n\;\rho_{n-1}\;(\mathtt{sd}+n-1) \\
\qquad\qquad \mathtt{code}_V\;e_0\;\rho_n\;(\mathtt{sd}+n) \\
\qquad\qquad \mathtt{slide\ n}
\end{array}
$$

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

CBV semantics uses $\mathtt{code}_V$ (and not $\mathtt{code}_C$) for evaluation of e_i.

NB! All expressions e_i have the same global environment.

Local definitions

Example: let $e \equiv \textbf{let}\ a = 19;\ b = a * a\ \textbf{in}\ a + b$ with environment $\rho = \emptyset$.

$\text{code}_V\ e\ \rho\ 0$ emits the following code under 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     2   getbasic      3   slide 2
```

Local definitions

CBN generates the following code:

$$\mathrm{code}_V\ (\mathbf{letrec}\ y_1 = e_1;\ \ldots;\ y_n = e_n\ \mathbf{in}\ e_0)\ \rho\ \mathsf{sd} \quad =$$

```
alloc n
```
$\mathrm{code}_C\ e_1\ \rho'\ (\mathsf{sd} + n)$
```
rewrite n
```
$\ldots$

$\mathrm{code}_C\ e_n\ \rho'\ (\mathsf{sd} + n)$
```
rewrite 1
```
$\mathrm{code}_V\ e_0\ \rho'\ (\mathsf{sd} + n)$
```
slide n
```

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

CBV semantics uses code_V (and not code_C) for evaluation of e_i.

NB! Under CBV, expressions e_i are not allowed to be primitive values.

Local definitions

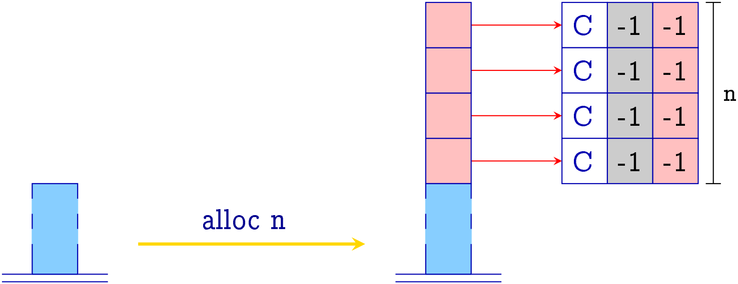
Example:

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

$\text{code}_V\ e\ \emptyset\ 0$ generates the following code (under CBV):

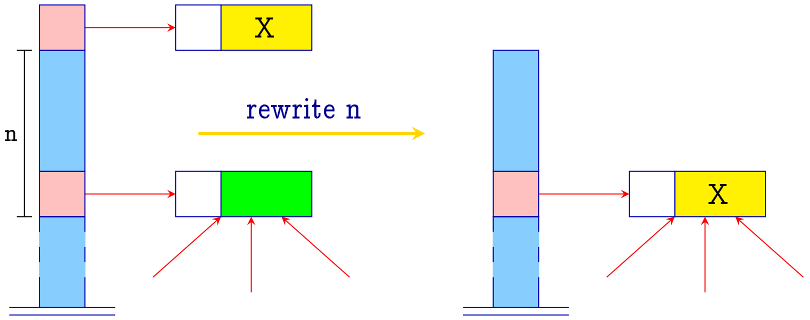
```
0       alloc 1
1       pushloc 0
2       mkvec 1
2       mkfunval A
2       jump B
0   A:  targ 2
0       ...
0       return 2
2   B:  rewrite 1
1       mark C
4       loadc 1
5       mkbasic
5       pushloc 4
6       apply
2   C:  slide 1
```

Local definitions



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

Local definitions



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

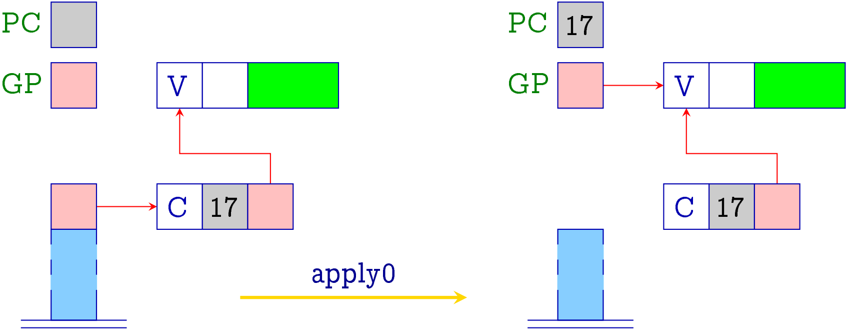
- The pointer `S[SP-n]` doesn't change!
- Only its *contents* is changed!

Closures

- Closure are necessary for CBN semantics.
- Before a variable is accessed, its value must be available.
- Otherwise, the closure it is bound must be evaluated.
- A closure is essentially a parameterless function.
- Hence, its evaluation is its application to 0 arguments.
- Evaluation of a closure is performed by the instruction `eval`.

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

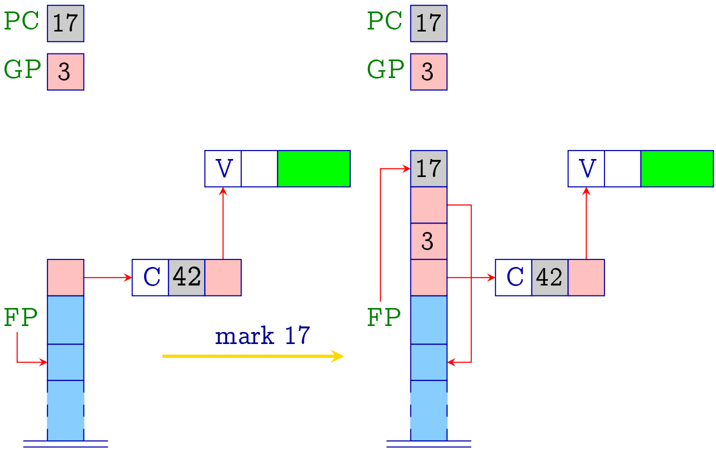
Closures



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

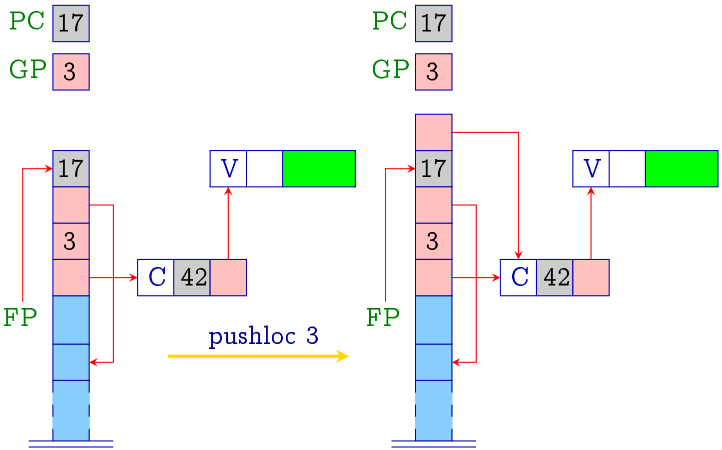
Closures

Evaluation of a closure by `eval`:



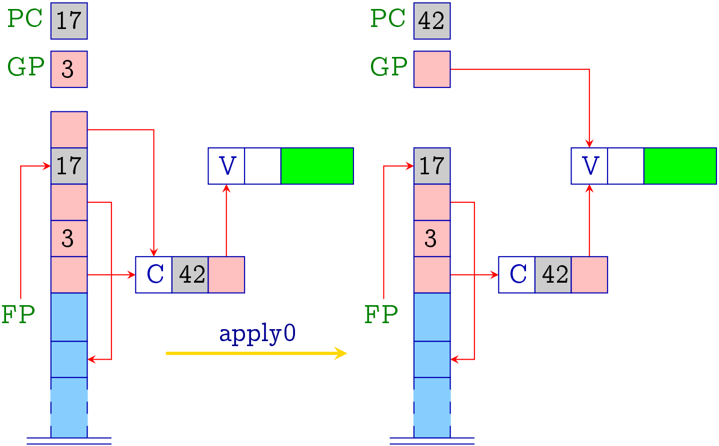
Closures

Evaluation of a closure by `eval`:



Closures

Evaluation of a closure by `eval`:



Closures

Construction of a closure for an expression e:

- packs its free variables into a global vector;
- creates a C-object which points to the global vector and to a start address of the code which evaluates the expression.

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

$$\begin{array}{llcl} \text{where} & \{z_0, \ldots, z_{g-1}\} & = & \mathit{free}(e) \\ & \rho' & = & \{z_i \mapsto (G, i) \mid i = 0, \ldots, g-1\} \end{array}$$

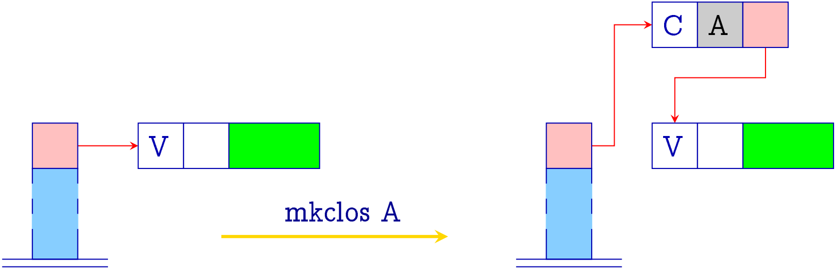
Closures

Example: let $e \equiv a * a$ with environment $\rho = \{a \mapsto (L, 0)\}$ and sd $= 1$.

$\text{code}_C\ e\ \rho$ sd generates the code:

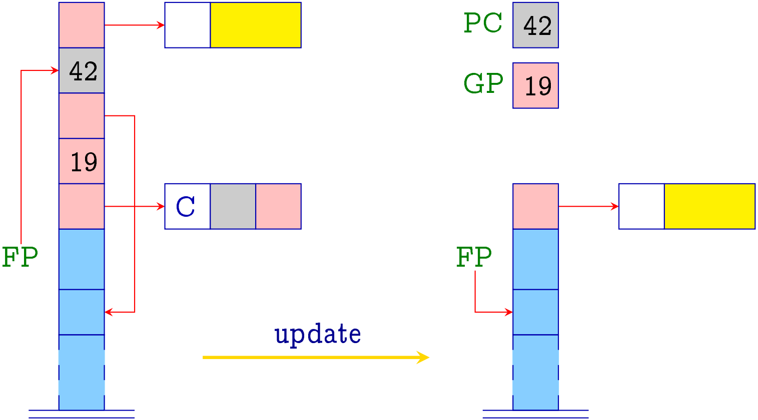
```
1    pushloc 1
2    mkvec 1
2    mkclos A
2    jump B
0    A: pushglob 0
1       eval
1       getbasic
2       pushglob 0
2       eval
2       getbasic
2       mul
1       mkbasic
1       update
2    B: ...
```

Closures



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

Closures

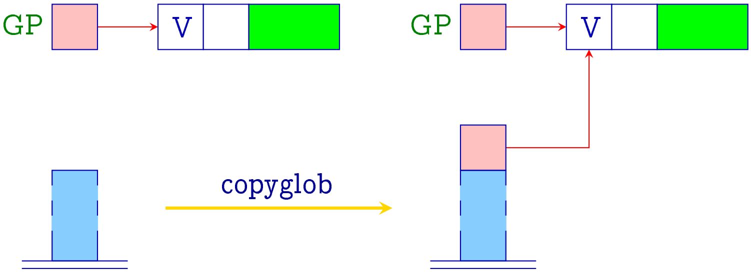


Optimization I: Global Variables

- Functional programs construct many F- and C-objects.
- In particular, this requires creation of global vectors.
- Top level variables can be statically bound to absolute addresses which can be used for their access.
  - Since these absolute addresses are known at compile-time, there is no need to add them to global vectors.
- Often it is also possible to reuse global vectors.
  - Useful, for instance, for compiling let-expressions or function applications, where one may construct a single global vector containing all free variables of definitions or arguments.

Optimization I: Global Variables

Similarly to local variables, reusable global variables are saved in the stack.



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

Optimization I: Global Variables

- Shared global vectors may contain more free variables than those in the given expression:
  - the more there are variables, the higher is a probability that one can reuse the vector.
- Unnecessary variables may lead to memory leaks.
- Possible solution: delete the reference after its ”life span”.

Optimization II: Closures

- Construction of a closure for expression e delays its evaluation until its value is really needed.
- If the value is not needed at all, the closure remains unevaluated (*lazy evaluation*).
- But if we know statically that the value is certainly needed (eg. by strictness analysis), the construction of a closure is wasted additional work.
- Hence, if expression e is in a *strict context*, then:

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

- Construction of a closure may also be unnecessary if the expression is very simple.

Optimization II: Closures

Primitive values:

Construction of a closure for primitive values is at least as expensive as direct construction of B-object!
Hence:

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

This replaces the code sequence:

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

Optimization II: Closures

Variables:

A variable is bound either to a value or a C-object, and construction of a new closure is unnecessary. Hence:

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

This replaces the code sequence:

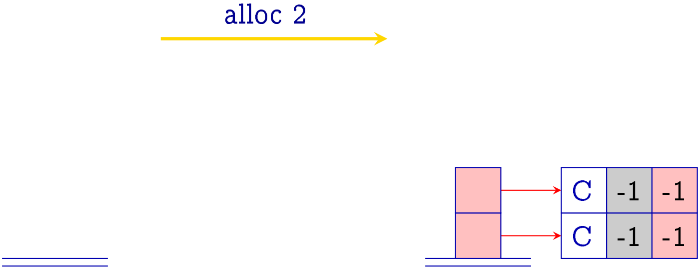
```
getvar x ρ sd     mkclos A     A: pushglob 0          update
mkvec 1           jump B          eval             B: ...
```

Example: let $e \equiv \textbf{letrec}\ a = b;\ b = 7\ \textbf{in}\ a$, then $\mathsf{code}_V\ e\ \emptyset\ 0$ generates:

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

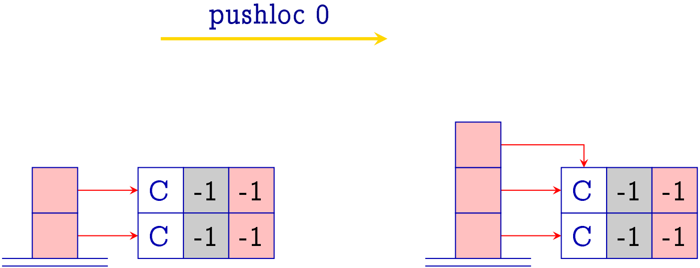
Optimization II: Closures

```
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
```



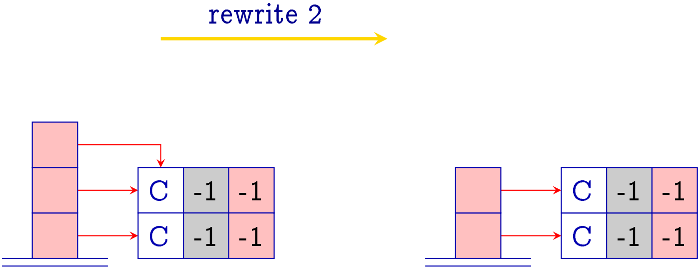
Optimization II: Closures

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



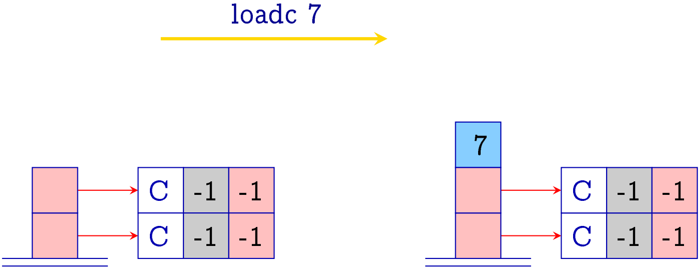
Optimization II: Closures

```
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
```



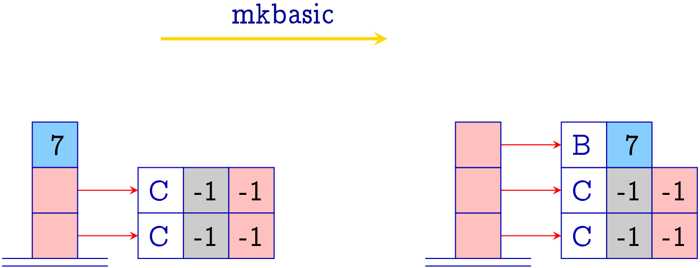
Optimization II: Closures

```
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
```



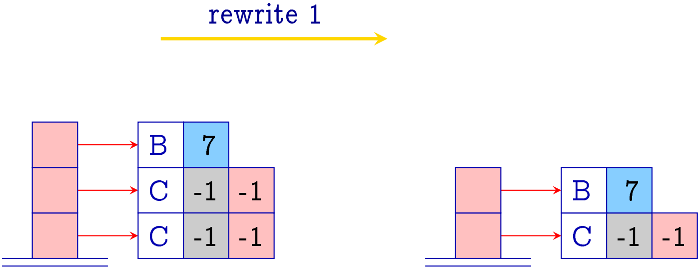
Optimization II: Closures

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



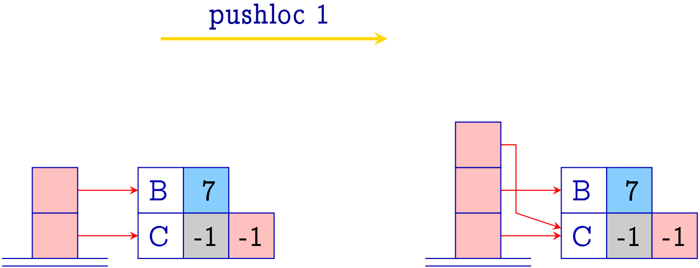
Optimization II: Closures

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



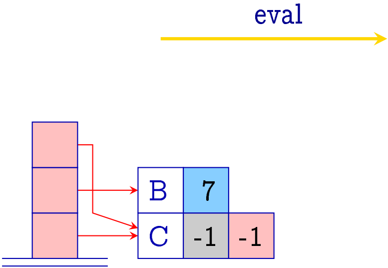
Optimization II: Closures

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



Optimization II: Closures

```
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!!

Optimization II: Closures

Seems that the optimization was not completely correct!

Problem:

Variable x was bound to the value of y before the later was replaced by the real value!!

Solution:

cyclic definitions: not to allow definitions of the form

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

acyclic definitions: reorder definitions by their dependency order.

Optimization II: Closures

Functions:

Functions are already values and can't be evaluated further. Instead of generating a code to create a closure for F-object, we can construct the F-object directly.

Hence:

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

Translation of a complete program

The initial state of the abstract machine:

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

Program (ie. expression) e can't contain any *free variables*.

Generated code will evaluate the expression e and then stops the machine using the instruction `halt`:

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

Translation of a complete program

- Given compilation schemes generate "*spaghetti code*".
- Reason: the code for function bodies and closures is placed directly after instructions `mkfunval` and `mkclos`, and then jumping over this code.
- Alternative: put this code somewhere else; eg. after instruction `halt`:

  Benefits: no need for jumps after `mkfunval` and `mkclos`.

  Drawbacks: compilation schemes become more complicated.
- Solution: eliminate the "spaghetti code" after the code generation by a special optimization phase.

Translation of a complete program

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

After elimination of the "spaghetti code" we get:

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

Data Structures

Extended PuF with data structures:

tuples:

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

lists:

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

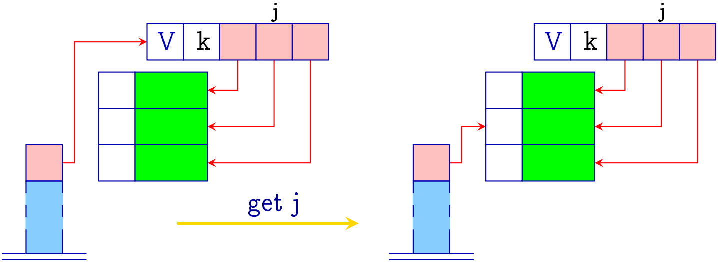
Data Structures

Construction of a tuple pushes its components into the stack and constructs a vector.
For selection of a component, the tuple is evaluates into a vector and the component with the corresponding index is returned.

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

Under CBV components are evaluated directly using code_V.

Data Structures



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

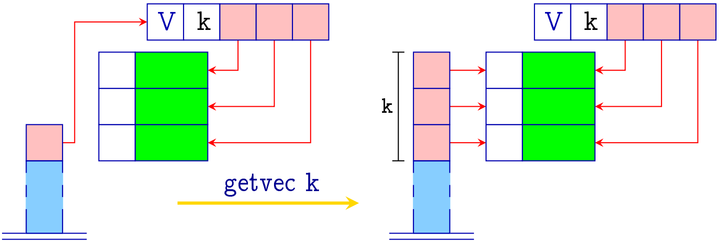
Data Structures

To access all components, the tuple is evaluated into vector and pointers to all its components are pushed into the stack.

$$\begin{array}{lll}
\mathrm{code}_V\ (\mathbf{let}\ (y_0,\ldots,y_{k-1}) = e_1\ \mathbf{in}\ e_0)\ \rho\ \mathrm{sd} & = & \mathrm{code}_V\ e_1\ \rho\ \mathrm{sd} \\
 & & \texttt{getvec k} \\
 & & \mathrm{code}_V\ e_0\ \rho'\ \mathrm{sd} \\
 & & \texttt{slide k}
\end{array}$$

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

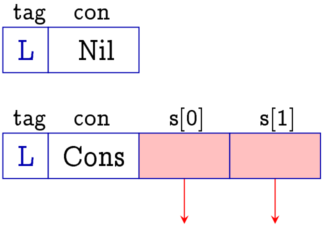
Data Structures



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

Data Structures

List constructors are represented by new kinds of objects:



tag con

L Nil

Empty List

tag con s[0] s[1]

L Cons

Non-empty List

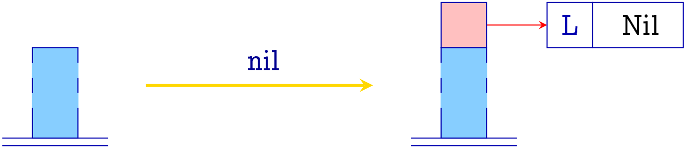
Data Structures

Construction of a list evaluates its arguments (if it has them; ie. in case of ":"), and creates a corresponding object in the heap:

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

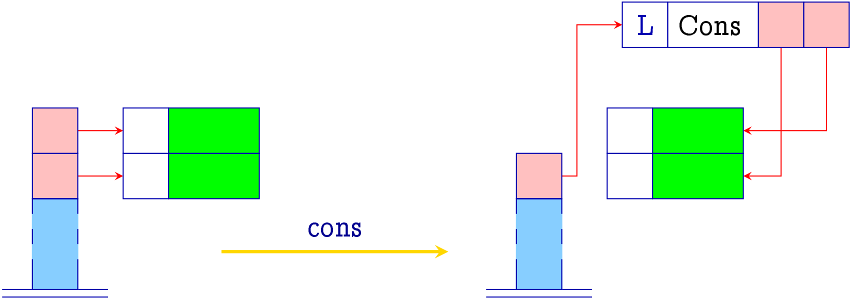
Under CBV, the head and tail are evaluated by code_V.

Data Structures



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

Data Structures



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

Data Structures

- Inspection of lists is performed by pattern matching.
- Evaluation of a case-expression
  $e \equiv \textbf{case } e_0 \textbf{ of } [\,] \to e_1;\ h : t \to e_2$:
  - evaluates an expression e_0;
  - if the value of e_0 is an empty list, evaluates the expression e_1;
  - if the value of e_0 is a non-empty list, then pushes the pointers to its head and tail into the stack (ie. binds variables h and t), and evaluates the expression e_2.

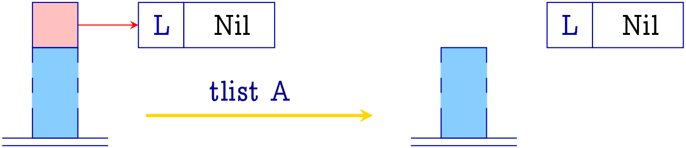
Data Structures

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

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

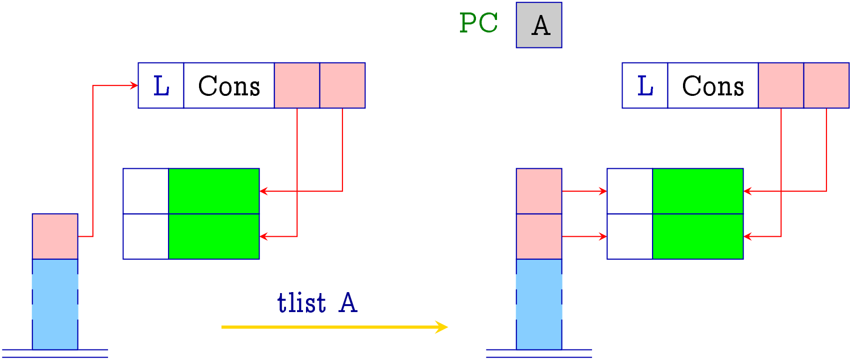
NB! Is the same for CBN and CBV.

Data Structures



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

Data Structures



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

Data Structures

Example:

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

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

Data Structures

If the tuple or list is in a closure context, there is no need to construct the closure but may construct the corresponding object directly:

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

Tail Recursion

- A function application is in a *tail position* if its value may be the value of the whole expression
  - the application $r\ t\ (h : y)$ is in a tail position in:

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

  - the application $f(x-1)$ is not in a tail position in:

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

- A function is *tail recursive* if all its recursive calls (both direct and indirect ones) are in tail positions.
- There is no need to create a new frame for the application in a tail position!

Tail Recursion

If the application $e' \; e_0 \; \ldots \; e_{m-1}$ is in a tail position, the generated code:

- binds formal parameters with arguments e_i and evaluates an expression e' to a F-object;
- deallocates local variables in the active frame;
- applies the function to its arguments.

NB! Evaluation of arguments and a function is done in the currently active frame.

Tail Recursion

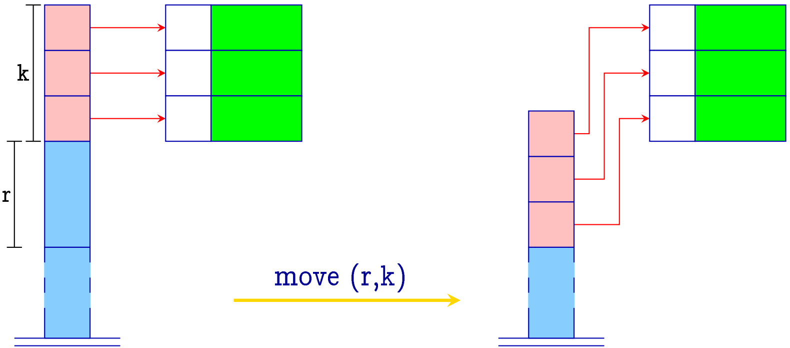
Under CBN the following code is generated:

$$
\begin{array}{lcl}
\mathrm{code}_V\ (e'\ e_0\ \ldots\ e_{m-1})\ \rho\ \mathrm{sd} & = & \mathrm{code}_C\ e_{m-1}\ \rho\ \mathrm{sd} \\
 & & \mathrm{code}_C\ e_{m-2}\ \rho\ (\mathrm{sd}+1) \\
 & & \ldots \\
 & & \mathrm{code}_C\ e_0\ \rho\ (\mathrm{sd}+m-1) \\
 & & \mathrm{code}_V\ e'\ \rho\ (\mathrm{sd}+m) \\
 & & \mathtt{move}\ (\mathrm{sd}+k,\ m+1) \\
 & & \mathtt{apply}
\end{array}
$$

where k is a number of parameters of the ”outer” function.

CBV uses code_V for evaluating arguments e_i (instead of code_C).

Tail Recursion



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

Tail Recursion

Example:

$$
rev = \mathbf{fn}\ x, y \Rightarrow \quad \begin{array}{lll} \multicolumn{3}{l}{\mathbf{case}\ x\ \mathbf{of}} \\ \ [\,] & \to & y \\ \ h : t & \to & rev\ t\ (h : y) \end{array}
$$

Under CBN the following code is generated for the body of *rev*:

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

0        jump B

2     A: pushloc 1
3        pushloc 4
4        cons
3        pushloc 1

4        pushglob 0
5        eval
5        move (4,3)
         apply

1     B: return 2
```

Since old organizational cells are still present, the instruction `return 2` is reachable only by direct jump from the branch corresponding to the empty list.