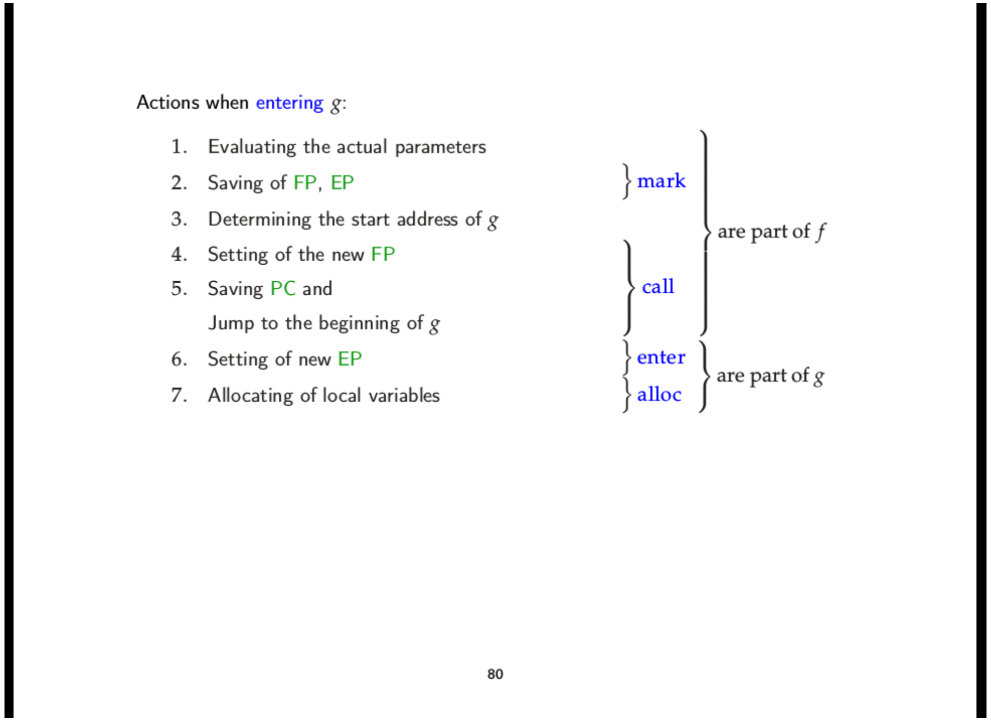
Script generated by TTT

Title: Seidl: Virtual_Machines (26.04.2016)

Date: Tue Apr 26 10:22:20 CEST 2016

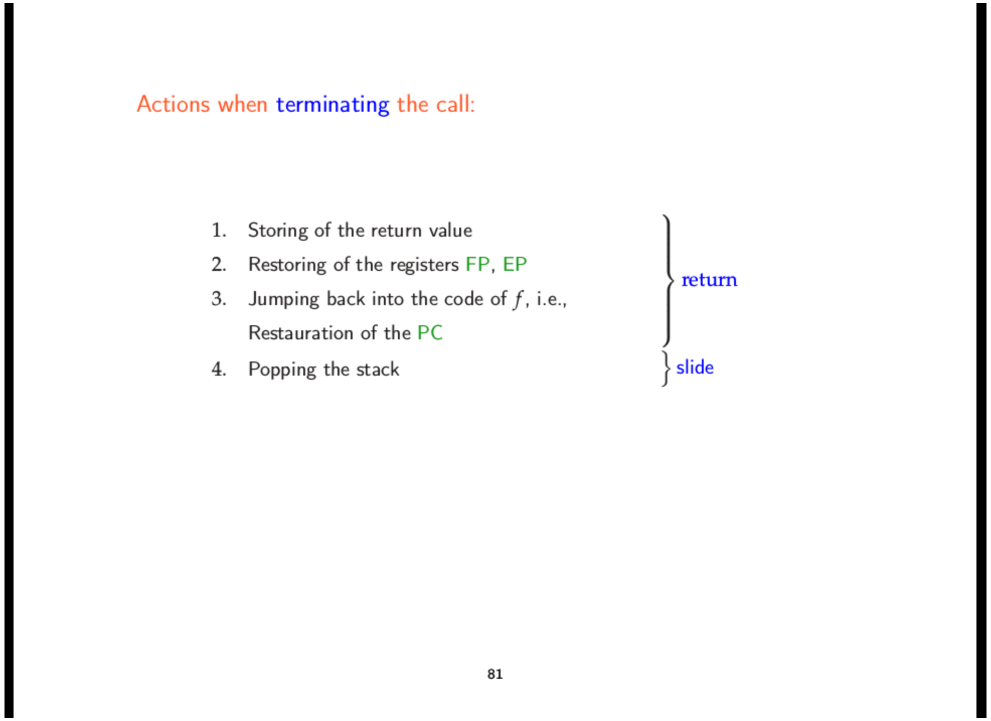
Duration: 91:14 min

Pages: 36



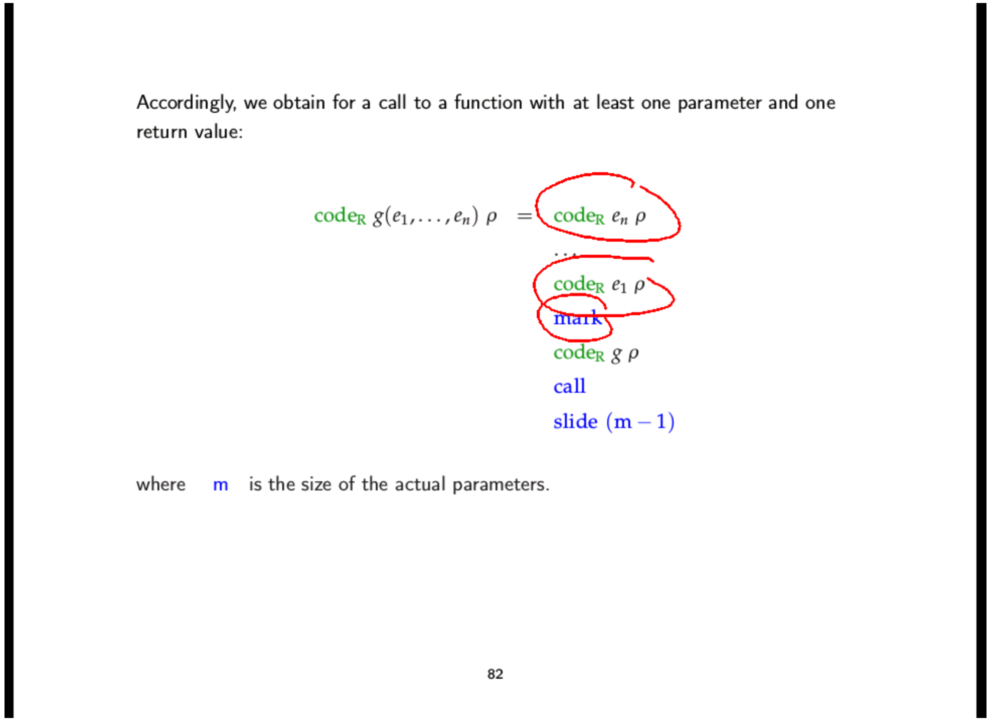
Actions when entering g:

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



Actions when terminating the call:

1.	Storing of the return value	return
2.	Restoring of the registers FP, EP	
3.	Jumping back into the code of f, i.e., Restauration of the PC	
4.	Popping the stack	slide



Accordingly, we obtain for a call to a function with at least one parameter and one return value:

$$\begin{array}{lll} \text{code}_R\ g(e_1,\ldots,e_n)\ \rho & = & \text{code}_R\ e_n\ \rho \\ & & \ldots \\ & & \text{code}_R\ e_1\ \rho \\ & & \text{mark} \\ & & \text{code}_R\ g\ \rho \\ & & \text{call} \\ & & \text{slide}\ (\text{m}-1) \end{array}$$

where m is the size of the actual parameters.

- Similar to declared arrays, function names are interpreted as constant pointes onto function code. Thus, the R-value of this pointer is the start address of the function.
- Caveat! For a variable **int** $(*)()\ g;$ the two calls

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

are equivalent! By means of normalization, the dereferencing of function pointers can be considered as redundant.

- During passing of parameters, these are copied.

Consequently,

$$\begin{array}{lll} \text{code}_{\text{R}}\ f\ \rho & = & \text{loadc}\ (\rho\ f) & \quad f \text{ name of a function} \\ \text{code}_{\text{R}}\ (*e)\ \rho & = & \text{code}_{\text{R}}\ e\ \rho & \quad e \text{ function pointer} \\ \text{code}_{\text{R}}\ e\ \rho & = & \text{code}_{\text{L}}\ e\ \rho & \\ & & \text{move k} & \quad e \text{ a structure of size } k \end{array}$$

where

Remark

- Of every expression which is passed as a parameter, we determine the R-value
  $\Longrightarrow$ call-by-value passing of parameters.
- The function g may as well be denoted by an expression, whose R-value provids the start address of the called function ...

- Similar to declared arrays, function names are interpreted as constant pointes onto function code. Thus, the R-value of this pointer is the start address of the function.
- Caveat! For a variable **int** $(*)()\ g;$ the two calls

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

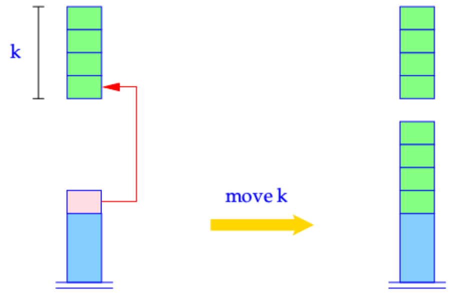
are equivalent! By means of normalization, the dereferencing of function pointers can be considered as redundant.

- During passing of parameters, these are copied.

Consequently,

$$\begin{array}{lll} \text{code}_{\text{R}}\ f\ \rho & = & \text{loadc}\ (\rho\ f) & \quad f \text{ name of a function} \\ \text{code}_{\text{R}}\ (*e)\ \rho & = & \text{code}_{\text{R}}\ e\ \rho & \quad e \text{ function pointer} \\ \text{code}_{\text{R}}\ e\ \rho & = & \text{code}_{\text{L}}\ e\ \rho & \\ & & \text{move k} & \quad e \text{ a structure of size } k \end{array}$$

where

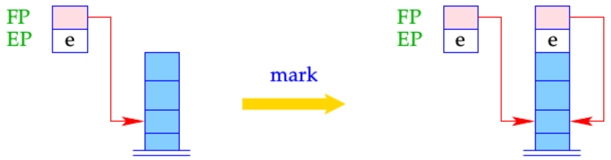


```
for (i = k-1; i≥0; i--)
        S[SP+i] = S[S[SP]+i];
SP = SP+k–1;
```

Remark

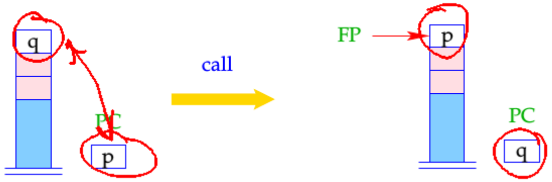
- Of every expression which is passed as a parameter, we determine the R-value
  ⟹ call-by-value passing of parameters.
- The function g may as well be denoted by an expression, whose R-value provids the start address of the called function ...

The instruction mark saves the registers FP and EP onto the stack.



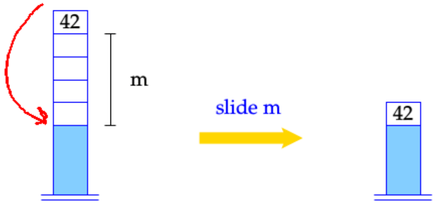
S[SP+1] = EP;
S[SP+2] = FP;
SP = SP + 2;

The instruction call saves the return address and sets FP and PC onto the new values.



tmp = S[SP];
S[SP] = PC;
FP = SP;
PC = tmp;

The instruction slide copies the return values into the correct memory cell:



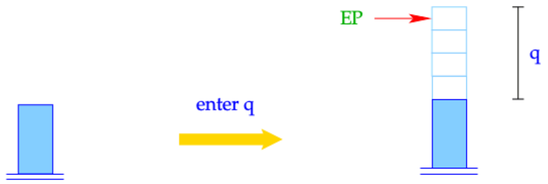
tmp = S[SP];
SP = SP-m;
S[SP] = tmp;

Accordingly, we translate a function definition:

```
code  t f (specs){V_defs  ss}  ρ  =
        _f:  enter q      // initialize EP
             alloc k      // allocate the local variables
             code ss ρ_f
             return       // return from call
```

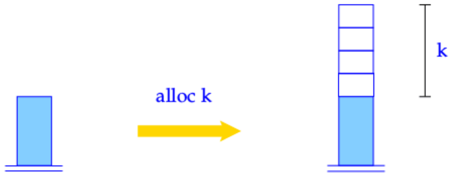
```
where  q    =  max + k          with
       max  =  maximal length of the local stack
       k    =  size of the local variables
       ρ_f  =  address environment for f
               //  takes specs, V_defs  and ρ into account
```

The instruction enter q sets the EP to the new value. If not enough space is available, program execution terminates.



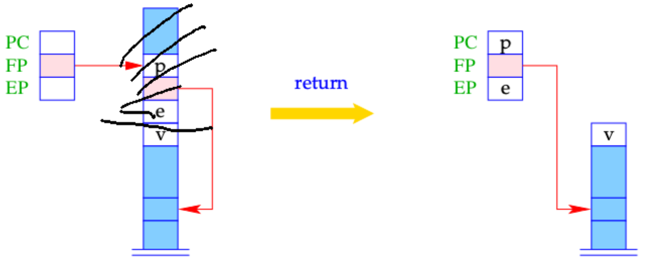
```
EP = SP + q;
if (EP ≥ NP)
    Error ("Stack Overflow");
```

The instruction alloc k allocates memory for locals on the stack.



```
SP = SP + k;
```

The instruction return pops the current stack frame. This means it restores the registers PC, EP and FP and returns the return value on top of the stack.



```
PC = S[FP]; EP = S[FP-2];
if (EP ≥ NP) Error ("Stack Overflow");
SP = FP-3; FP = S[SP+2];
```

9.4 Access to Variables, Formal Parameters and Returning of Values

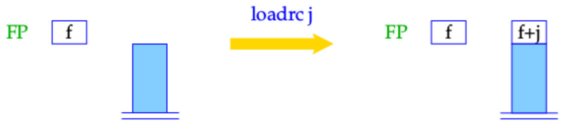
Accesses to local variables or formal parameters are relative to the current FP.

Accordingly, we modify code_L for names of variables.

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

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

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



SP++;
S[SP] = FP+j;

As an optimization, we introduce analogously to loada j and storea j the new instructions loadr j and storer j :

$$\begin{array}{lll} \text{loadr j} & = & \text{loadrc j} \\ & & \text{load} \\ \\ \text{storer j} & = & \text{loadrc j;} \\ & & \text{store} \end{array}$$

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

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

Example For 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 -3             mul
           alloc 0      storer -3  1        loadr -3             storer -3
2          loadr -3     return              loadc 1    3         return
           loadc 0      jump B              sub             B:   return
           leq                              mark       - 4
           jumpz A                          loadc _fac   5
                                            call
                                            slide 0    - 5
```

where $\rho_{\text{fac}} : x \mapsto (L, -3)$ and $q = 5$.

10 Translation of Whole Programs

Before program execution, we have:

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

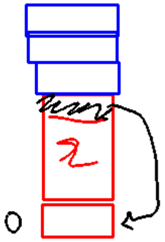
Let $p \equiv V\_defs \;\; F\_def_1 \dots F\_def_n$, denote a program where F_def_i is the definition of a function f_i of which one is called main .

The code for the program p consists of:

- code for the function definitions F_def_i;
- code for the allocation of global variables;
- code for the call of int main();
- the instruction halt which returns control to the operating system together with the value at address 0.

Then we define:

$$
\begin{array}{rcll}
\text{code}\; p\; \emptyset & = & & \text{enter } (k+4) \\
 & & & \text{alloc } (k+1) \\
 & & & \text{mark} \\
 & & & \text{loadc \_main} \\
 & & & \text{call} \\
 & & & \text{slide } k \\
 & & & \text{halt} \\
 & & \_f_1: & \text{code } F\_def_1\; \rho \\
 & & & \vdots \\
 & & \_f_n: & \text{code } F\_def_n\; \rho
\end{array}
$$



where $\emptyset \;\widehat{=}\;$ empty address environment;
$\rho \;\widehat{=}\;$ global address environment;
$k \;\widehat{=}\;$ size of the global variables

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;
- Exceptions.

A program is an expression e of the form:

$$
\begin{array}{rcl}
e & ::= & b \mid x \mid (\Box_1\, e) \mid (e_1\, \Box_2\, e_2) \\
 & \mid & (\mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2) \\
 & \mid & (e'\, e_0 \ldots e_{k-1}) \\
 & \mid & (\mathbf{fun}\ x_0 \ldots x_{k-1} \to e) \\
 & \mid & (\mathbf{let}\ x_1 = e_1\ \mathbf{in}\ e_0) \\
 & \mid & (\mathbf{let\ rec}\ x_1 = e_1\ \mathbf{and} \ldots \mathbf{and}\ x_n = e_n\ \mathbf{in}\ e_0)
\end{array}
$$

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 **let-rec**-expression, i.e. an expression with simultaneously defined local variables.

For simplicity, we only allow `int` as basic type.

Example

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

$$
\begin{array}{ll}
\mathbf{let\ rec}\ \mathrm{fac} = & \mathbf{fun}\ x \to \mathbf{if}\ x \leq 1\ \mathbf{then}\ 1 \\
 & \mathbf{else}\ x \cdot \mathrm{fac}\ (x-1) \\
\mathbf{in}\ \mathrm{fac}\ 7 &
\end{array}
$$

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).

let rec fac x = if . . . . .
in fac 7

A program is an expression e of the form:

$$
\begin{array}{rcl}
e & ::= & b \mid x \mid (\Box_1\, e) \mid (e_1\, \Box_2\, e_2) \\
 & \mid & (\mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2) \\
 & \mid & (e'\, e_0 \ldots e_{k-1}) \\
 & \mid & (\mathbf{fun}\ x_0 \ldots x_{k-1} \to e) \\
 & \mid & (\mathbf{let}\ x_1 = e_1\ \mathbf{in}\ e_0) \\
 & \mid & (\mathbf{let\ rec}\ x_1 = e_1\ \mathbf{and} \ldots \mathbf{and}\ x_n = e_n\ \mathbf{in}\ e_0)
\end{array}
$$

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 **let-rec**-expression, i.e. an expression with simultaneously defined local variables.

For simplicity, we only allow `int` as basic type.

Example

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

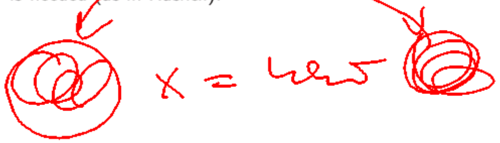
$$\begin{aligned} \textbf{let rec } \mathsf{fac} &= \textbf{fun } x \to \textbf{if } x \leq 1 \textbf{ then } 1 \\ &\quad\; \textbf{else } x \cdot \mathsf{fac}\ (x-1) \\ \textbf{in } \mathsf{fac}\ 7 & \end{aligned}$$

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).



Example

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

$$\begin{aligned} \textbf{let rec } \mathsf{fac} &= \textbf{fun } x \to \textbf{if } x \leq 1 \textbf{ then } 1 \\ &\quad\; \textbf{else } x \cdot \mathsf{fac}\ (x-1) \\ \textbf{in } \mathsf{fac}\ 7 & \end{aligned}$$

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

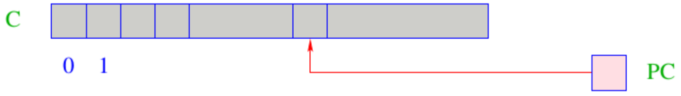
There are two Semantics:

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).

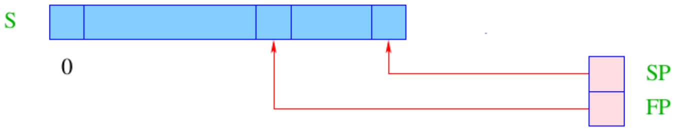
12 Architecture of the MaMa

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;

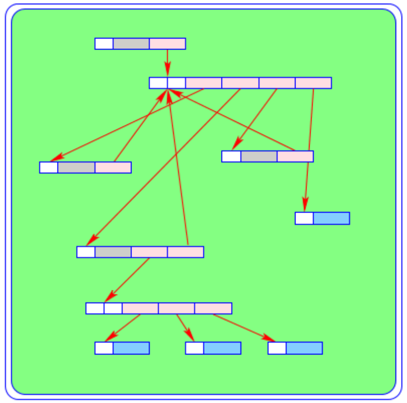


S = Runtime-Stack – each cell can hold a basic value or an address;

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:



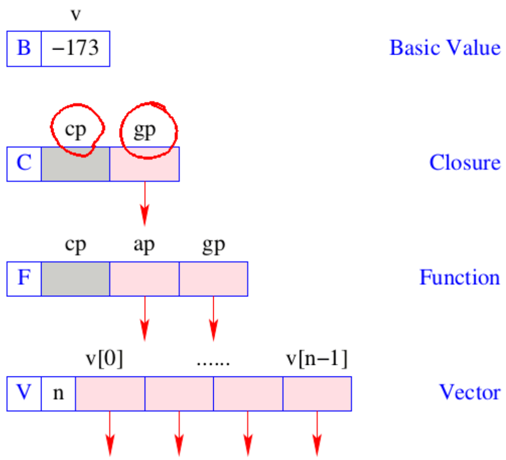
□ Tag

▨ Code Pointer

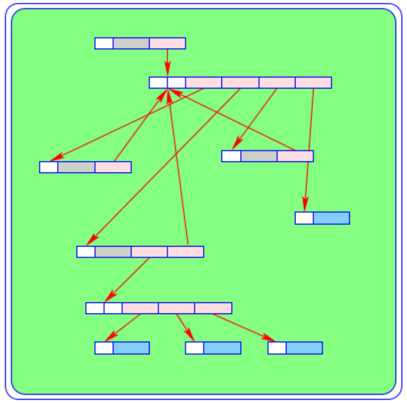
🟦 Value

▭ Heap Pointer

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



We also need a heap H:



□ Tag

▨ Code Pointer

🟦 Value

▭ Heap Pointer

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:

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: