Script generated by TTT

Title: Petter: Virtual Machines (14.05.2019)

Date: Tue May 14 10:15:22 CEST 2019

Duration: 96:20 min

Pages: 27

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,

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,

A Slightly Larger Example

$$\textbf{let } a = 17 \textbf{ in let } f = \textbf{fun } b \to a + b \textbf{ in } f\ 42$$

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	6	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

A Slightly Larger Example

let $a = 17$ **in let** $f =$ **fun** $b \to a + b$ **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	6		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

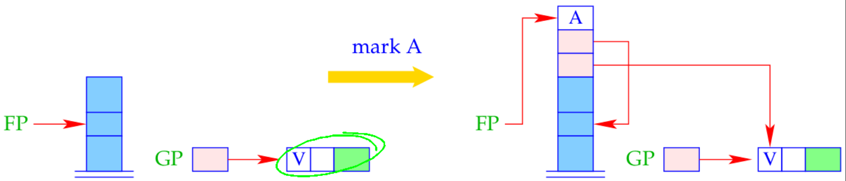
17 Function Application

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.

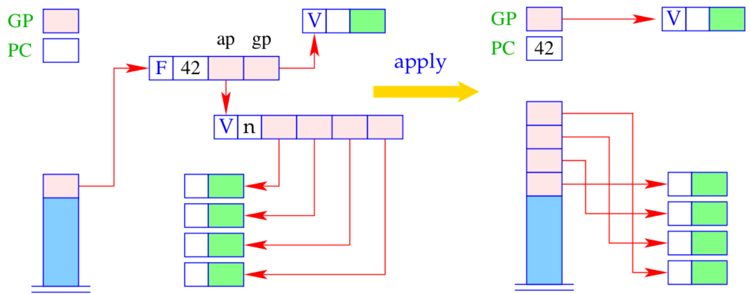
Thus for CBN,

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



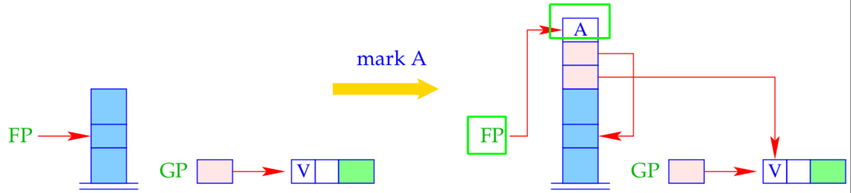
S[SP+1] = GP;
S[SP+2] = FP;
S[SP+3] = A;
FP = SP = SP + 3;

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:



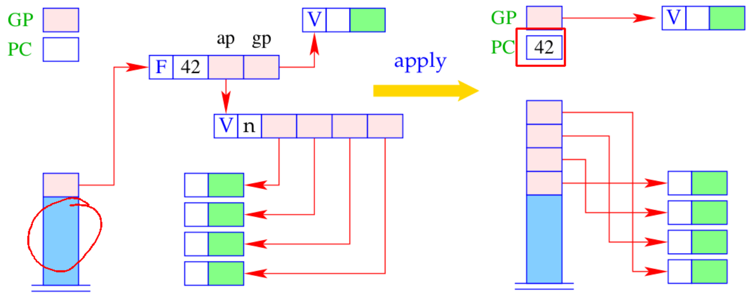

```
h = S[SP];
if (H[h] != (F,_,_))
    Error "no fun";
else {
    GP = h→gp; PC = h→cp;
    for (i=0; i¡ h→ap→n; i++)
        S[SP+i] = h→ap→v[i];
    SP = SP + h→ap→n – 1;
}
```

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



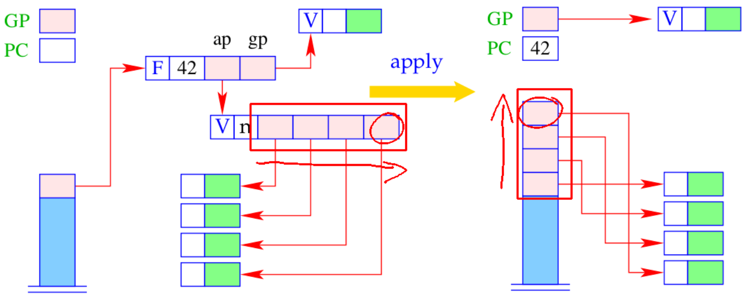

```
S[SP+1] = GP;
S[SP+2] = FP;
S[SP+3] = A;
FP = SP = SP + 3;
```


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;
}
```


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;
}
```


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.

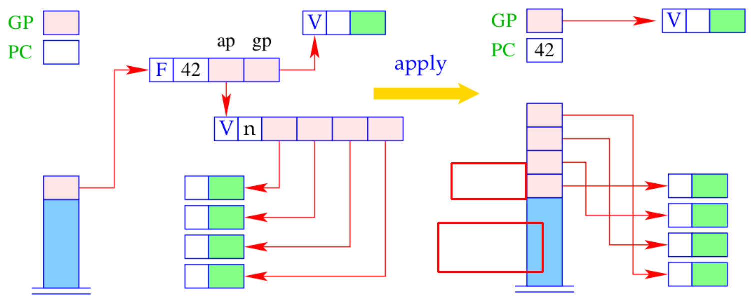
This instruction checks whether there are enough arguments to evaluate the body.

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

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];                    GP = h→gp; PC = h→cp;
if (H[h] != (F,_,_))          for (i=0; i¡ h→ap→n; i++)
    Error "no fun";               S[SP+i] = h→ap→v[i];
else {                        SP = SP + h→ap→n – 1;
                              }
```


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 .

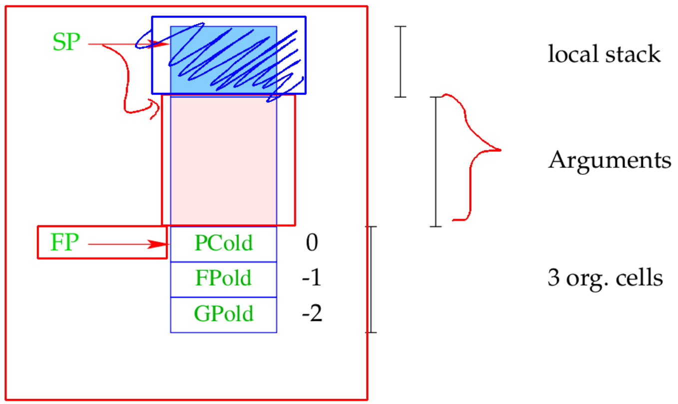
This instruction checks whether there are enough arguments to evaluate the body.

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

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



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

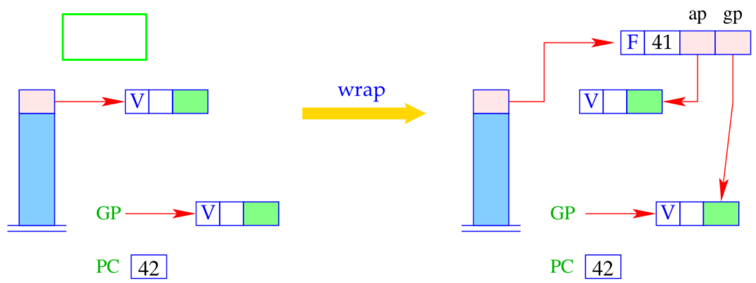
targ k is a complex instruction.

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
           }
```

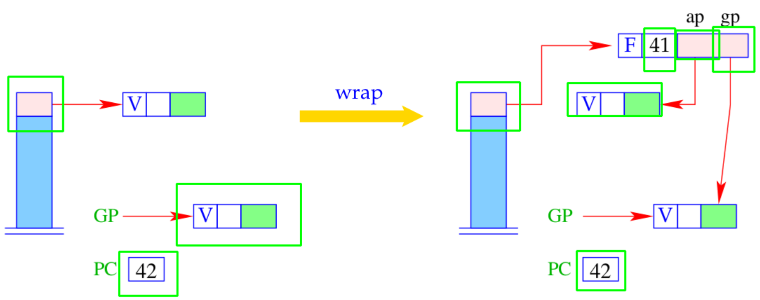
The combination of these steps into one instruction is a kind of optimization.

The instruction wrap wraps the argument vector together with the global vector and PC-1 into an F-object:



S[SP] = new (F, PC-1, S[SP], GP);

The instruction wrap wraps the argument vector together with the global vector and PC-1 into an F-object:



S[SP] = new (F, PC-1, S[SP], GP);

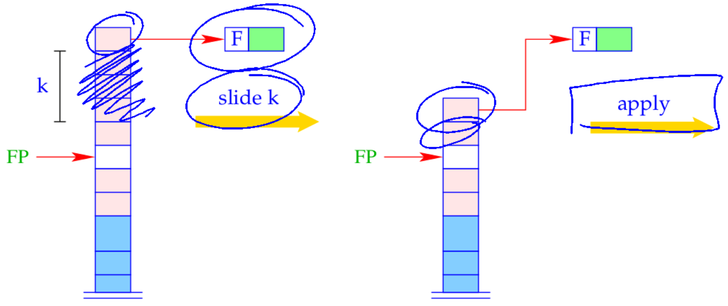
targ k is a complex instruction.

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.

Case: Over-supply



19 let-rec-Expressions

Consider the expression $e \equiv \textbf{let rec } y_1 = e_1 \textbf{ and} \ldots \textbf{and } y_n = e_n \textbf{ in } e_0$.

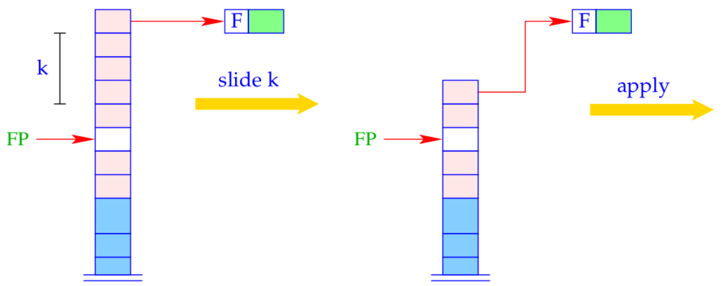
The translation of e must deliver an instruction sequence that

- 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.

Caveat

In a **let-rec** expression, the definitions can use variables that will be allocated only later! $\Longrightarrow$ Dummy-values are put onto the stack before processing the definition.

Case: Over-supply



19 let-rec-Expressions

Consider the expression $e \equiv \textbf{let rec } y_1 = e_1 \textbf{ and} \ldots \textbf{and } y_n = e_n \textbf{ in } e_0$.

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.

Caveat

In a **let-rec** 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 for $e \equiv \textbf{let rec } y_1 = e_1 \textbf{ and} \ldots \textbf{and } y_n = e_n \textbf{ in } e_0$:

$\text{code}_V\ e\ \rho\ \text{sd} \quad = \quad$ alloc n $\qquad$ // allocates local variables

$\qquad \text{code}_C\ e_1\ \rho'\ (\text{sd}+n)$

$\qquad$ rewrite n

$\qquad \ldots$

$\qquad \text{code}_C\ e_n\ \rho'\ (\text{sd}+n)$

$\qquad$ rewrite 1

$\qquad \text{code}_V\ e_0\ \rho'\ (\text{sd}+n)$

$\qquad$ slide n $\qquad$ // de-allocates local variables

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

In the case of CBV, we also use code_V for the expressions $e_1, \ldots, e_n$.

Caveat

Recursive definitions of basic values are undefined with CBV!!!

Example

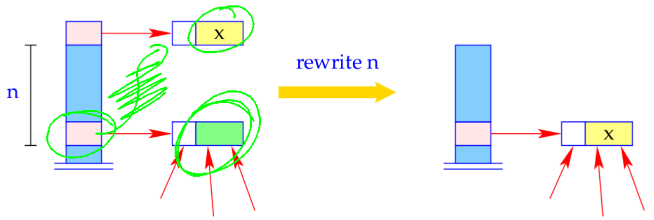
Consider the expression

$$e \equiv \textbf{let rec } f = \textbf{fun } x\ y \rightarrow \textbf{if } y \leq 1 \textbf{ then } x \textbf{ else } f(x*y)(y-1) \textbf{ in } f\ 1$$

for $\rho = \emptyset$ and $\text{sd} = 0$. 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 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 in the implementation of CBN for **let**-, **let-rec** expressions as well as for actual paramaters of functions.
- 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.