

Title: Petter: Virtual Machines (27.05.2019)

Date: Mon May 27 10:19:28 CEST 2019

Duration: 85:52 min

Pages: 8

The code for a last call $l \equiv (e' e_0 \dots e_{m-1})$ 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. de-allocate the local variables and the k consumed arguments of f ;
3. execute an `apply`.

```

codev l ρ sd = codec em-1 ρ sd
              codec em-2 ρ (sd + 1)
              ...
              codec e0 ρ (sd + m - 1)
              codev e' ρ (sd + m) // evaluation of the function
              move r (m + 1) // de-allocation of r cells
              apply
    
```

where $r = \boxed{sd + k}$ is the number of stack cells to de-allocate.

25 Last Calls

A function application is called **last call** in an expression e if this application could deliver the value for e .

A recursive function definition is called **tail recursive** if all recursive calls are last calls.

Examples

$r t (h :: y)$ is a **last call** in

```

match x with
[] → y
| h :: t → r t (h :: y)
    
```

$f (x - 1)$ is **not a last call** in

```

if x ≤ 1
then 1
else x * f (x - 1)
    
```

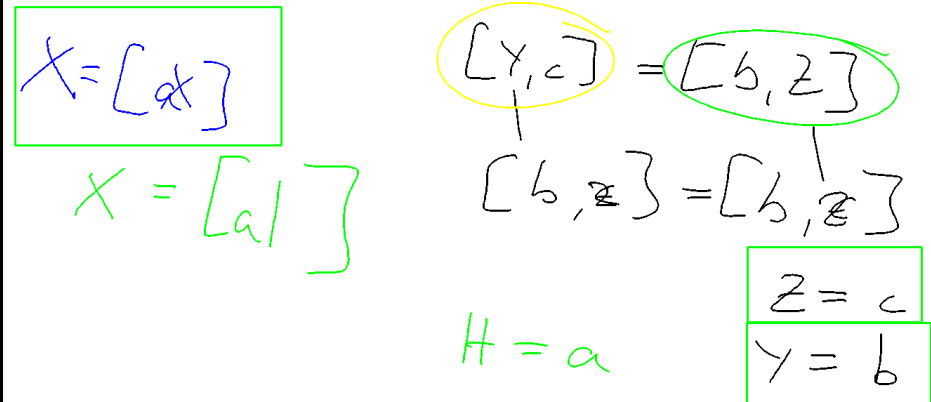
Observation: Last calls in a function body need **no new** stack frame!

⇒ Automatic transformation of tail recursion into loops!!!

A More Realistic Example

```

app(X, Y, Z) ← X = [], Y = Z
app(X, Y, Z) ← X = [H|X], Z = [H|Z], app(X, Y, Z)
? app[X, [Y, c], [a, b, Z]]
    
```



Procedural View of Proll programs

literal	==	procedure call
predicate	==	procedure
clause	==	definition
term	==	value
unification	==	basic computation step
binding of variables	==	side effect

Note: Predicate calls ...

- ... do not have a return value.
- ... affect the caller through side effects only.
- ... may fail. Then the next definition is tried.

⇒ backtracking

234

A program p is constructed as follows:

$$\begin{aligned}
 t &::= a \mid X \mid - \mid f(t_1, \dots, t_n) \\
 g &::= p(t_1, \dots, t_k) \mid X = t \\
 c &::= p(X_1, \dots, X_k) \leftarrow g_1, \dots, g_r \\
 p &::= c_1 \dots c_m ? g
 \end{aligned}$$

- A **term** t either is an atom, a variable, an anonymous variable or a constructor application.
- A **goal** g either is a literal, i.e., a predicate call, or a unification.
- A **clause** c consists of a **head** $p(X_1, \dots, X_k)$ with predicate name and list of formal parameters together with a **body**, i.e., a sequence of goals.
- A **program** consists of a sequence of clauses together with a single goal as **query**.

233

Procedural View of Proll programs

literal	==	procedure call
predicate	==	procedure
clause	==	definition
term	==	value
unification	==	basic computation step
binding of variables	==	side effect

Note: Predicate calls ...

- ... do not have a return value.
- ... affect the caller through side effects only.
- ... may fail. Then the next definition is tried.

⇒ backtracking

234

A program p is constructed as follows:

$$\begin{aligned}
 t &::= a \mid X \mid - \mid f(t_1, \dots, t_n) \\
 g &::= p(t_1, \dots, t_k) \mid X = t \\
 c &::= p(X_1, \dots, X_k) \leftarrow g_1, \dots, g_r \\
 p &::= c_1 \dots c_m ? g
 \end{aligned}$$

- A **term** t either is an atom, a variable, an anonymous variable or a constructor application.
- A **goal** g either is a literal, i.e., a predicate call, or a unification.
- A **clause** c consists of a **head** $p(X_1, \dots, X_k)$ with predicate name and list of formal parameters together with a **body**, i.e., a sequence of goals.
- A **program** consists of a sequence of clauses together with a single goal as **query**.

233

Procedural View of Proll programs

literal	==	procedure call
predicate	==	procedure
clause	==	definition
term	==	value
unification	==	basic computation step
binding of variables	==	side effect

Note: Predicate calls ...

- ... do not have a return value.
- ... affect the caller through side effects only.
- ... may fail. Then the next definition is tried.

⇒ backtracking