Script **generated by TTT**

Title: Seidl: Virtual_Machines (22.05.2013)

Date: Wed May 22 16:00:38 CEST 2013

Duration: 90:44 min

Pages: 34

---

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 $\textbf{match}\ x\ \textbf{with}\ [\,]\ \to y \mid h::t\ \to r\ t\ (h::y)$

$f\ (x-1)$ 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.

$$\begin{array}{llll}
\text{code}_V\ l\ \rho\ \boxed{\text{sd}} & = & \text{code}_C\ e_{m-1}\ \rho\ \text{sd} & \\
 & & \text{code}_C\ e_{m-2}\ \rho\ (\text{sd}+1) & \\
 & & \ldots & \\
 & & \text{code}_C\ e_0\ \rho\ (\text{sd}+m-1) & \\
 & & \text{code}_V\ e'\ \rho\ (\text{sd}+m) & //\ \text{Evaluation of the function} \\
 & & \text{move}\ r\ (m+1) & //\ \text{Deallocation of } r \text{ cells} \\
 & & \text{apply} &
\end{array}$$

where $r = sd + k$ is the number of stack cells to deallocate.

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.

$$\begin{array}{llll}
\text{code}_V\ l\ \rho\ \text{sd} & = & \text{code}_C\ e_{m-1}\ \rho\ \text{sd} & \\
 & & \text{code}_C\ e_{m-2}\ \rho\ (\text{sd}+1) & \\
 & & \ldots & \\
 & & \text{code}_C\ e_0\ \rho\ (\text{sd}+m-1) & \\
 & & \text{code}_V\ e'\ \rho\ (\text{sd}+m) & //\ \text{Evaluation of the function} \\
 & & \text{move}\ r\ (m+1) & //\ \text{Deallocation of } r \text{ cells} \\
 & & \text{apply} &
\end{array}$$

where $r = sd + k$ is the number of stack cells to deallocate.

$\rho = \{ r \mapsto (G, 0),\ x \mapsto (L, 0),\ y \mapsto (L, -1) \}$

Example:

The body of the function

$$r = \mathbf{fun}\ x\ y\ \rightarrow\ \mathbf{match}\ x\ \mathbf{with}\ [\,]\ \rightarrow y \mid h :: t\ \rightarrow r\ t\ (h :: y)$$

```
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.

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.

$$\begin{aligned}
\text{code}_V\ l\ \rho\ \text{sd} = \ & \text{code}_C\ e_{m-1}\ \rho\ \text{sd} \\
& \text{code}_C\ e_{m-2}\ \rho\ (\text{sd}+1) \\
& \ldots \\
& \text{code}_C\ e_0\ \rho\ (\text{sd}+m-1) \\
& \text{code}_V\ e'\ \rho\ (\text{sd}+m) && \text{// Evaluation of the function} \\
& \text{move}\ r\ (m+1) && \text{// Deallocation of } r \text{ cells} \\
& \text{apply}
\end{aligned}$$

where $r = sd + k$ is the number of stack cells to deallocate.

$-(m+1)$

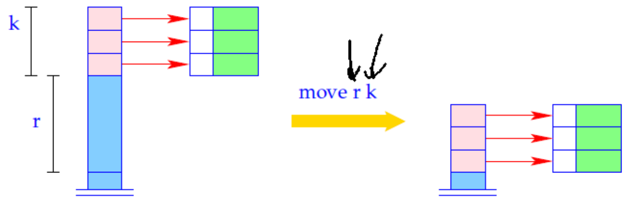
Example:

The body of the function

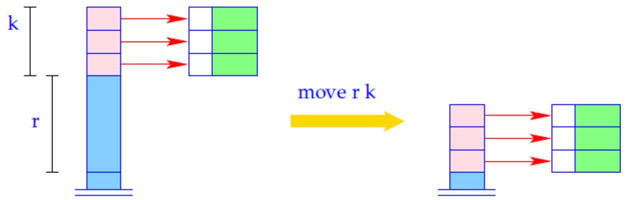
$$r = \mathbf{fun}\ x\ y\ \rightarrow\ \mathbf{match}\ x\ \mathbf{with}\ [\,]\ \rightarrow y \mid h :: t\ \rightarrow r\ t\ (h :: y)$$

```
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
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;
```



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

The Translation of Logic Languages

---

26 The Language Proll

Here, we just consider the core language Proll (“Prolog-light” :-). In particular, we omit:

- arithmetic;
- the cut operator;
- self-modification of programs through assert and retract.

Example:

$\text{bigger}(X,Y) \leftarrow X = elephant, Y = horse$.
$\text{bigger}(X,Y) \leftarrow X = horse, Y = donkey$,
$\text{bigger}(X,Y) \leftarrow X = donkey, Y = dog$,
$\text{bigger}(X,Y) \leftarrow X = donkey, Y = monkey$,
$\text{is\_bigger}(X,Y) \leftarrow \text{bigger}(X,Y)$.
$\text{is\_bigger}(X,Y) \leftarrow \text{bigger}(X,Z), \text{is\_bigger}(Z,Y)$.
?- $\text{is\_bigger}(elephant, dog)$.

Example:

$\forall X, Y.$

$\text{bigger}(X,Y) \leftarrow X = \mathit{elephant}, Y = \mathit{horse}$
$\text{bigger}(X,Y) \leftarrow X = \mathit{horse}, Y = \mathit{donkey}$
$\text{bigger}(X,Y) \leftarrow X = \mathit{donkey}, Y = \mathit{dog}$
$\text{bigger}(X,Y) \leftarrow X = \mathit{donkey}, Y = \mathit{monkey}$
$\text{is\_bigger}(X,Y) \leftarrow \text{bigger}(X,Y)$
$\text{is\_bigger}(X,Y) \leftarrow \text{bigger}(X,Z), \text{is\_bigger}(Z,Y)$
$?\ \text{is\_bigger}(\mathit{elephant}, \mathit{dog})$

A More Realistic Example:

$\text{app}(X,Y,Z) \leftarrow X = [\,],\ Y = Z$
$\text{app}(X,Y,Z) \leftarrow X = [H|X'],\ Z = [H|Z'],\ \text{app}(X',Y,Z')$
$?\ \text{app}(X, [Y,c], [a,b,Z])$

↑ ↑ ↑

$X = a$
$Y = b$
$Z = c$

A More Realistic Example:

$\text{app}(X,Y,Z) \leftarrow X = [\,],\ Y = Z$
$\text{app}(X,Y,Z) \leftarrow X = [H|X'],\ Z = [H|Z'],\ \text{app}(X',Y,Z')$
$?\ \text{app}(X, [Y,c], [a,b,Z])$

Remark:

$[\,]$	$==$	the atom empty list
$[H|Z]$	$==$	binary constructor application
$[a,b,Z]$	$==$	shortcut for: $[a|[b|[Z|[\,]]]]$

A program p is constructed as follows:

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

- 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 $p(X_1,\ldots,X_k)$ 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.

A More Realistic Example:

$$
\begin{array}{lll}
\mathsf{app}(X,Y,Z) & \leftarrow & X = [\,],\ Y = Z \\
\mathsf{app}(X,Y,Z) & \leftarrow & X = [H|X'],\ Z = [H|Z'],\ \mathsf{app}(X',Y,Z') \\
? & \mathsf{app}(X,[Y,c],[a,b,Z]) &
\end{array}
$$

Remark:

$[\,]$	$==$	the atom empty list
$[H|Z]$	$==$	binary constructor application
$[a,b,Z]$	$==$	shortcut for: $[a|[b|[Z|[\,]]]]$

A program p is constructed as follows:

$$
\begin{array}{lcl}
t & ::= & a \mid X \mid \_ \mid f(t_1,\ldots,t_n) \\
g & ::= & p(t_1,\ldots,t_k) \mid X = t \\
c & ::= & p(X_1,\ldots,X_k) \leftarrow g_1,\ldots,g_r \\
p & ::= & c_1.\ldots.c_m?g
\end{array}
$$

- 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 $p(X_1,\ldots,X_k)$ 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.

$f(X,Y) = f(a, g(X))$

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])

Remark:

[ ] == the atom empty list
[H|Z] == binary constructor application
[a,b,Z] == shortcut for: [a|[b|[Z|[ ]]]]

$X = a,\ Y = g(X)$

A program p is constructed as follows:

t ::= a | X | _ | f(t_1,...,t_n)
g ::= p(t_1,...,t_k) | X = t
c ::= p(X_1,...,X_k) ← g_1,...,g_r
p ::= c_1.....c_m?g

- 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,...,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.

Procedural View of Proll programs:

goal == 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 :-))

$\Longrightarrow$ backtracking

A program p is constructed as follows:

$$
\begin{array}{lcl}
t & ::= & a \mid X \mid \_ \mid f(t_1, \ldots, t_n) \\
g & ::= & p(t_1, \ldots, t_k) \mid X = t \\
c & ::= & p(X_1, \ldots, X_k) \leftarrow \boxed{g_1, \ldots, g_r} \\
p & ::= & c_1. \ldots .c_m ?g
\end{array}
$$

- 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 $p(X_1, \ldots, X_k)$ 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.

Procedural View of Proll programs:

goal == 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 :-))

$\Longrightarrow$ backtracking

A program p is constructed as follows:

$$
\begin{array}{lcl}
t & ::= & a \mid X \mid \_ \mid f(t_1, \ldots, t_n) \\
g & ::= & p(t_1, \ldots, t_k) \mid X = t \\
c & ::= & p(X_1, \ldots, X_k) \leftarrow g_1, \ldots, \overset{\downarrow}{g_r} \\
p & ::= & c_1. \ldots .c_m ?g
\end{array}
$$

- 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 $p(X_1, \ldots, X_k)$ 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.

Procedural View of Proll programs:

goal == 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 :-))

$\Longrightarrow$ backtracking

A More Realistic Example:

$$
\begin{array}{lll}
\mathsf{app}(X,Y,Z) & \leftarrow & X = [\,],\ Y = Z \\
\mathsf{app}(X,Y,Z) & \leftarrow & X = [H|X'],\ Z = [H|Z'],\ \mathsf{app}(X',Y,Z') \\
? \ \ \mathsf{app}(X,[Y,c],[a,b,Z]) & &
\end{array}
$$

Remark:

$[\,]$ == the atom empty list
$[H|Z]$ == binary constructor application
$[a,b,Z]$ == shortcut for: $[a|[b|[Z|[\,]]]]$

Procedural View of Proll programs:

goal == 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 :-))

$\Longrightarrow$ backtracking

A More Realistic Example:

$$
\begin{array}{lll}
\mathsf{app}(X,Y,Z) & \leftarrow & X = [\,],\ Y = Z \\
\mathsf{app}(X,Y,Z) & \leftarrow & X = [H|X'],\ Z = [H|Z'],\ \mathsf{app}(X',Y,Z') \\
? \ \ \mathsf{app}(X,[Y,c],[a,b,Z]) & &
\end{array}
$$

Remark:

$[\,]$ == the atom empty list
$[H|Z]$ == binary constructor application
$[a,b,Z]$ == shortcut for: $[a|[b|[Z|[\,]]]]$

Procedural View of Proll programs:

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

Note: Predicate calls ...

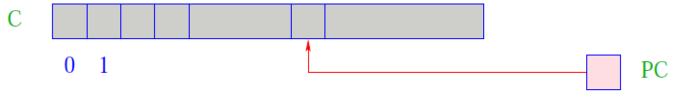
- ... do not have a return value.
- ... affect the caller through side effects only :-)
- ... may fail. Then the next definition is tried :-))

$\Longrightarrow$ backtracking

[illegible] [illegible]

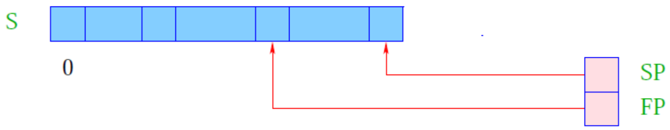
27 Architecture of the WiM:

The Code Store:



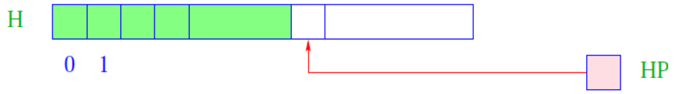
C = Code store – contains WiM program;
every cell contains one instruction;
PC = Program Counter – points to the next instruction to executed;

The Runtime Stack:



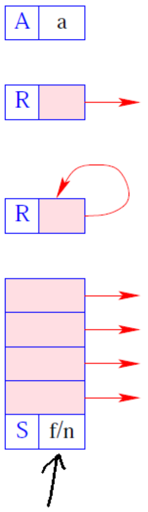
S = Runtime Stack – every cell may contain a value or an address;
SP = Stack Pointer – points to the topmost occupied cell;
FP = Frame Pointer – points to the current stack frame.
Frames are created for predicate calls,
contain cells for each variable of the current clause

The Heap:

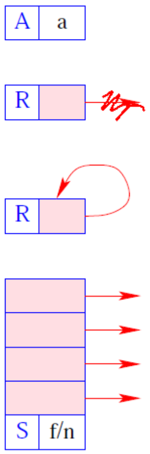


H = Heap for dynamicly constructed terms;
HP = Heap-Pointer – points to the first free cell;

- The heap in maintained like a stack as well :-)
- A `new`-instruction allocates a object in H.
- Objects are tagged with their types (as in the MaMa) ...



atom	1 cell
variable	1 cell
unbound variable	1 cell
structure	(n+1) cells



atom	1 cell
variable	1 cell
unbound variable	1 cell
structure	(n+1) cells

---

28 Construction of Terms in the Heap

Parameter terms of goals (calls) are constructed in the heap before passing.

Assume that the address environment ρ returns, for each clause variable X its address (relative to FP) on the stack. Then $\text{code}_A\ t\ \rho$ should ...

- construct (a presentation of) t in the heap; and
- return a reference to it on top of the stack.

Idea:

- Construct the tree during a post-order traversal of t
- with one instruction for each new node!

Example: $t \equiv f(g(X, Y), a, Z)$.

Assume that X is initialized, i.e., $S[FP + \rho\, X]$ contains already a reference, Y and Z are not yet initialized.