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:

\[ \text{rt (h :: y) is a last call in} \text{ match x with} \]
\[ \text{if } x \leq 1 \text{ then 1 else } x + f(x - 1) \]

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

Automatic transformation of tail recursion into loops!!!

The code for a last call \( l = (e' e_2 \ldots e_m) \) 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.

\[
\text{code}(l)_{(sd)} = \text{code}_{e_m-1} \rho \; \text{sd} \\
\text{code}_{e_m-2} \rho \; (\text{sd} + 1) \\
\vdots \\
\text{code}_{e_0} \rho \; (\text{sd} + m - 1) \\
\text{code}_{e'} \rho \; (\text{sd} + m) \quad // \text{Evaluation of the function} \\
\text{move r } (m + 1) \quad // \text{Deallocation of r cells} \\
\text{apply} \\
\]

where \( r = sd + k \) is the number of stack cells to deallocate.

The code for a last call \( l = (e' e_2 \ldots e_m) \) 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.

\[
\text{code}(l)_{\rho \; \text{sd}} = \text{code}_{e_m-1} \rho \; \text{sd} \\
\text{code}_{e_m-2} \rho \; (\text{sd} + 1) \\
\vdots \\
\text{code}_{e_0} \rho \; (\text{sd} + m - 1) \\
\text{code}_{e'} \rho \; (\text{sd} + m) \quad // \text{Evaluation of the function} \\
\text{move r } (m + 1) \quad // \text{Deallocation of r cells} \\
\text{apply} \\
\]

where \( r = sd + k \) is the number of stack cells to deallocate.
Example:

The body of the function

\[ r = \text{fun} x y \rightarrow \text{match} x \text{ with } [] \rightarrow y \mid h \cdot t \rightarrow r \cdot (h \cdot y) \]

Since the old stack frame is kept, return 2 will only be reached by the direct jump at the end of the []-alternative.

Example:

The body of the function

\[ r = \text{fun} x y \rightarrow \text{match} x \text{ with } [] \rightarrow y \mid h \cdot t \rightarrow r \cdot (h \cdot y) \]

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 \( l = (e' \; e_2 \ldots e_m) \) 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{align*}
\text{codev } l \cdot \text{sd} &= \text{codev } e_{m-1} \cdot \rho \cdot \text{sd} \\
\text{codev } e_{m-2} \cdot \rho \cdot (\text{sd } + 1) \\
\cdots \\
\text{codev } e_0 \cdot \rho \cdot (\text{sd } + m - 1) \\
\text{codev } e' \cdot \rho \cdot (\text{sd } + m) & \quad \quad \text{// Evaluation of the function} \\
\text{move } r \cdot (m + 1) & \quad \quad \text{// Deallocation of \( r \) cells} \\
\text{apply} \\
\end{align*}
\]

where \( r = (\text{sd } + k) \) is the number of stack cells to deallocate.
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:

- `bigger(X, Y) ← X = elephant, Y = horse .`
- `bigger(X, Y) ← X = horse, Y = donkey .`
- `bigger(X, Y) ← X = donkey, Y = dog .`
- `bigger(X, Y) ← X = donkey, Y = monkey .`
- `is_bigger(X, Y) ← bigger(X, Y) .`
- `is_bigger(X, Y) ← bigger(X, Z), is_bigger(Z, Y) .`
- `is_bigger(elephant, dog) .`
Example:

\[
\forall X, Y.
\begin{align*}
\text{bigger}(X, Y) & \leftarrow X = \text{elephant}, Y = \text{horse} \\
\text{bigger}(X, Y) & \leftarrow X = \text{horse}, Y = \text{donkey} \\
\text{bigger}(X, Y) & \leftarrow X = \text{donkey}, Y = \text{dog} \\
\text{bigger}(X, Y) & \leftarrow X = \text{donkey}, Y = \text{monkey} \\
\text{is\_bigger}(X, Y) & \leftarrow \text{bigger}(X, Y) \\
\text{is\_bigger}(X, Y) & \leftarrow \text{bigger}(X, Z), \text{is\_bigger}(Z, Y) \\
? \text{ is\_bigger(\text{elephant}, \text{dog})}
\end{align*}
\]

A More Realistic Example:

\[
\begin{align*}
\text{app}(X, Y, Z) & \leftarrow X = [\cdot], Y = Z \\
\text{app}(X, Y, Z) & \leftarrow X = [H[X]], Z = [H[Z]], \text{app}(X', Y, Z') \\
? \text{ app}(X, [Y, c], [a, b, Z])
\end{align*}
\]

Remark:

\[
\begin{align*}
[\cdot] & \quad \text{the atom empty list} \\
[H[Z]] & \quad \text{binary constructor application} \\
[a, b, Z] & \quad \text{shortcut for: } [a][b][Z]([])
\end{align*}
\]

A More Realistic Example:

\[
\begin{align*}
\text{app}(X, Y, Z) & \leftarrow X = [\cdot], Y = Z \\
\text{app}(X, Y, Z) & \leftarrow X = [H[X]], Z = [H[Z]], \text{app}(X', Y, Z') \\
? \text{ app}(X, [Y, c], [a, b, Z])
\end{align*}
\]

\[
\begin{align*}
X &= a \\
Y &= b \\
Z &= c
\end{align*}
\]

A program \( p \) is constructed as follows:

\[
\begin{align*}
\text{t} & \quad ::= \quad a \mid X \mid f(t_1, \ldots, t_n) \\
\text{g} & \quad ::= \quad p(t_1, \ldots, t_n) \mid X = t \\
\text{c} & \quad ::= \quad p(X_1, \ldots, X_n) \leftarrow g_1, \ldots, g_n \\
\text{p} & \quad ::= \quad c_1, \ldots, c_n \cdot \text{g}
\end{align*}
\]

- A term \( t \) either is an atom, a variable, an anonymous variable or a constructor application.
- A goal \( \text{g} \) either is a literal, i.e., a predicate call, or a unification.
- A clause \( \text{c} \) consists of a head \( p(X_1, \ldots, X_n) \) 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:

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

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:

\[
\begin{align*}
  t & := a \mid X \mid \_ \mid f(t_1, \ldots, t_n) \\
  g & := p(t_1, \ldots, t_n) \mid X = t \\
  c & := p(X_1, \ldots, X_n) \leftarrow g_1, \ldots, g_n \\
  p & := c_1 \cdots c_m, g
\end{align*}
\]

- 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 \( p(X_1, \ldots, X_n) \) 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.

\[
\begin{align*}
  f(X, Y) = f(a, g(X))
\end{align*}
\]

A More Realistic Example:

\[
\begin{align*}
  \text{app}(X, Y, Z) & \leftarrow X = [], \, Y = Z \\
  \text{app}(X, Y, Z) & \leftarrow X = [H|X'], \, Z = [H|Z'], \, \text{app}(X', Y, Z') \\
? & \quad \text{app}(X, Y, [c], [a, b, Z])
\end{align*}
\]

Remark:

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

\[
\begin{align*}
  X = a, \, Y = g(X)
\end{align*}
\]

A program \( p \) is constructed as follows:

\[
\begin{align*}
  t & := a \mid X \mid \_ \mid f(t_1, \ldots, t_n) \\
  g & := p(t_1, \ldots, t_n) \mid X = t \\
  c & := p(X_1, \ldots, X_n) \leftarrow g_1, \ldots, g_n \\
  p & := c_1 \cdots c_m, g
\end{align*}
\]

- 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 \( p(X_1, \ldots, X_n) \) 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 Prolog 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.
  => backtracking

A program $p$ is constructed as follows:

$$
t ::= a | X | f(t_1, \ldots, t_n)
g ::= p(t_1, \ldots, t_n) | X = t
c ::= p(X_1, \ldots, X_n) \leftarrow g_1, \ldots, g_n
\quad p ::= c_1 \ldots c_m$$

- 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 $p(X_1, \ldots, X_n)$ with predicate name and list of formal parameters together with a body, i.e., a sequence of goals.
- A program $p$ consists of a sequence of clauses together with a single goal as query.

- 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 $p(X_1, \ldots, X_n)$ with predicate name and list of formal parameters together with a body, i.e., a sequence of goals.
- A program $p$ consists of a sequence of clauses together with a single goal as query.
Procedural View of Prolog 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. (⇒)

⇒ backtracking

A More Realistic Example:

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

Remark:

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

Procedural View of Prolog 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. (⇒)

⇒ backtracking

A More Realistic Example:

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

Remark:

- [] === the atom empty list
- [H|Z] === binary constructor application
- [a, b, Z] === shortcut for: [a|b|[Z|[]]]
Procedural View of Prolog 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. backtrack

27 Architecture of the WiM:

The Code Store:

C 0 1  \[\text{Code store - contains WiM program; every cell contains one instruction;}
PC  \[\text{Program Counter - points to the next instruction to executed;}

The Runtime Stack:

S 0  \[\text{Runtime Stack - every cell may contain a value or an address;}
SP  \[\text{Stack Pointer - points to the topmost occupied cell;}
FP  \[\text{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 0 1  \[\text{Heap for dynamically constructed terms;}
HP  \[\text{Heap-Pointer - points to the first free cell;}

- The heap is maintained like a stack as well.
- A new-instruction allocates an object in H.
- Objects are tagged with their types (as in the MaMa) ...
28 Construction of Terms in the Heap

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

Assume that the address environment $\rho$ 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: $t = f(g(X, Y), a, Z)$.
Assume that $X$ is initialized, i.e., $S[FP + \rho X]$ contains already a reference, $Y$ and $Z$ are not yet initialized.