Handling Procedures

Mooly Sagiv
Tel Aviv University
sagiv@math.tau.ac.il
and
Reinhard Wilhelm
Universität des Saarlandes
wilhelm@cs.uni-sb.de

April 29, 1997

Subjects

- Scoping
- Static vs. Dynamic Binding
- Calling Trees
- Static Predecessor Tree
- Parameter Passing Mechanisms
- The Run-Time Stack Frame
- Addressing of Variables
- Computing the Address Environment
- Procedures as Parameters

Procedure Incarnations

- Calling a procedure p creates an *incarnation* $\hat{p}$
- An incarnation of a procedure contains incarnations of the formal parameters and the local variables
- Control

  1. enters the incarnation $\hat{p}$ from the caller $\hat{q}$,
  2. may pass to a procedure incarnation $\hat{r}$ created by a call,
  3. returns from $\hat{r}$,
  4. returns from $\hat{p}$ to $\hat{q}$.

Calling Tree

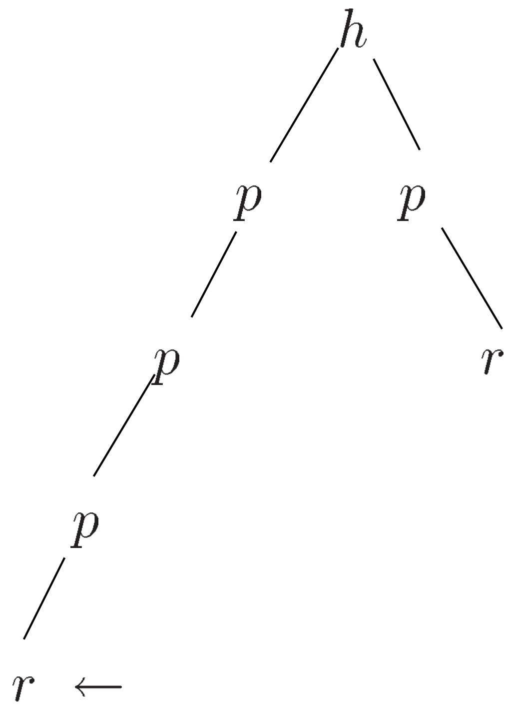
- Defined for an execution sequence
- An ordered tree
- The root is the main program
- There is a node for every procedure incarnation
- If p **calls** $p_1, p_2, \ldots, p_n$, then incarnations of the $p_1, p_2, \ldots, p_n$ are the children of p

A path starting with the root is called an **incarnation path**.

Example: Calling Tree

```
program h;
    var i : integer;
    proc p
        proc r

        if i > 0
            then i := i - 1; p
            else r
        fi
    i := 2;
    p;
    p
end.
```



A program and its calling tree

Static vs. Dynamic Binding

Binding strategy (*scope rules*) relates *applied* occurrences of names to *defining* occurrences.

Static: Applied occurrence of x denotes the defining occurrence of x in the next enclosing scope,

Dynamic: Applied occurrence of x denotes the last created incarnation of x.

```
var x
proc p
    proc q
        var x
        p
    x
    q
p
```

Static Binding

```
proc p(a)
    var b;
    var c
    proc q
        var a
        var q
        proc r
            var b
            b
            a
            c

        ⋮

        a
        b
        q
    proc s
        var a
        ⋮
        a
        q
    a
    q
```

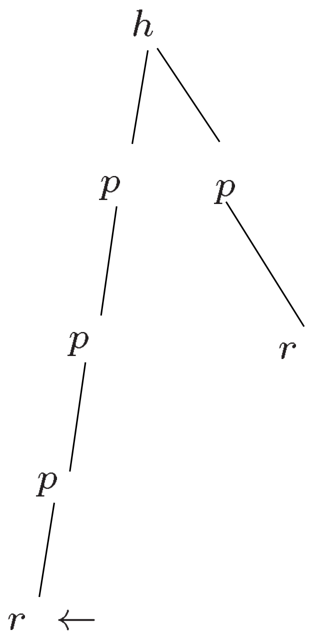
Static Predecessor Tree

- Defined for an incarnation path of a call tree
- Its root is the main program
- There is a node for every procedure incarnation of the path
- $\hat{p}$ is parent of $\hat{q}$, iff q is declared directly inside p and $\hat{p}$ is the first incarnation of p "above" $\hat{q}$ in the path

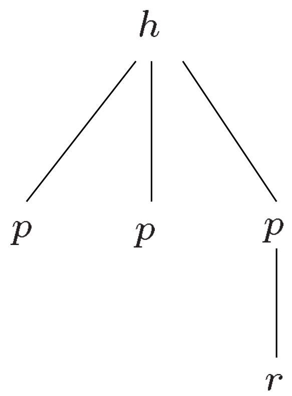
Warning! In general only true for programs without formal procedures.

Example: Static Predecessor Trees

```
program h;
    var i : integer;
    proc p
        proc r
        
        if i > 0
            then i := i − 1; p
            else r
        fi
    i := 2;
    p;
    p
end.
```



A program and its calling tree



Its static predecessor tree

Parameter Passing Mechanisms

value: The r-value of the actual parameter is passed.

value-result: Only variable actual parameters allowed.
Their r-value is passed to and returned from the callee.

var/reference: Only variable actual parameters allowed.
Their l-value is passed to the callee.

name: Actual parameter is evaluated every time execution needs it.
"Thunk"" (pointer to evaluation code and pointer to caller's frame) is passed.

Example

program test ;
var i : **integer** ;
procedure p(. . . x, y: **integer**);
begin
    while $x > 0$ **do**
        $x := x - 1$;
        $y := y + 1$
    end
end
begin
    $i := 3$;
    $p(i, i)$
end

. . . = value:

. . . = var:

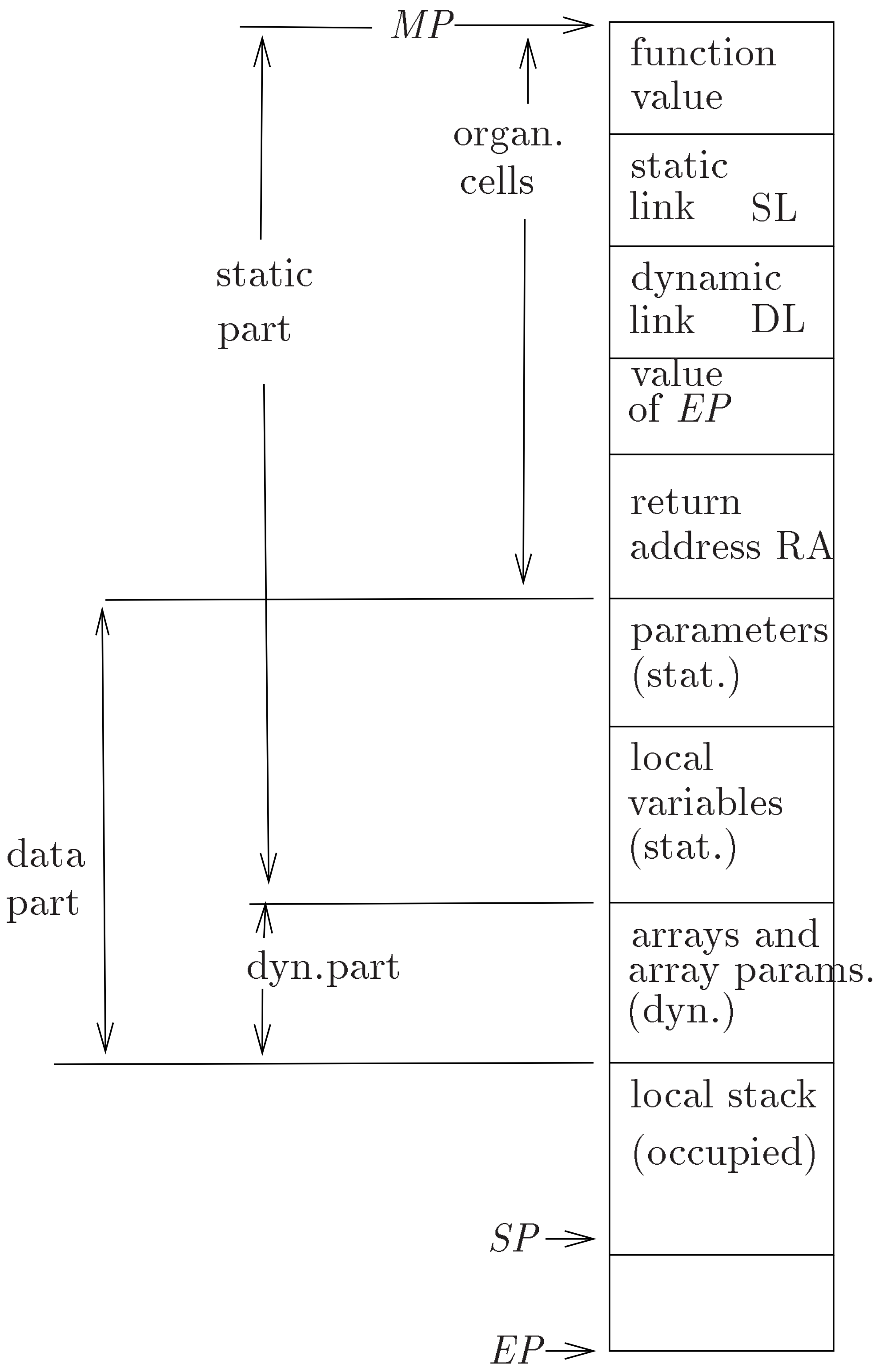
. . . = value-result:

Implementing Recursion

- Traversing the call tree with a nonrecursive program,
- needs a stack,
- elements in the stack are nodes of the call tree (i.e. procedure incarnations),
- stack content is prefix of an incarnation path (starting at the root),
- structure of this stack is isomorphic to the *run-time stack* for implementing procedures.

The Structure of an Activation Record

The representation of a procedure incarnation



Further outline

- Conceptual view of addressing
- Adapt memory allocation scheme ρ
- Adapt code for l-values
- Code for function/procedure calls/return
- Code for parameter passing
- Code for the function/procedure body

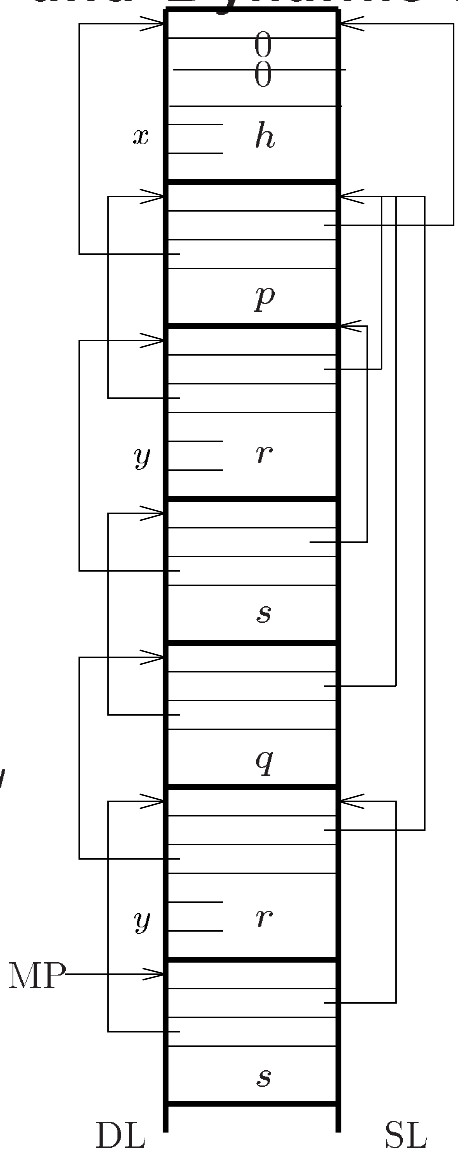
Addressing Global Variables

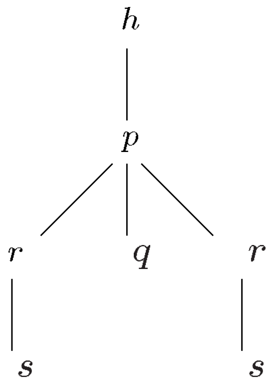
Using *nesting depth* of name occurrences
Inductively defined

- $nd(main) = 0$
- Types, variables, and procedures declared inside procedure p have nesting depth $nd(p) + 1$
- Applied occurrences of names inside procedure p have nesting depth $nd(p) + 1$

Static and Dynamic Links

```
program h;
    var x
    proc p
        ⋮
        proc q
            ⋮
            r
            ⋮
        proc r
            var y
            proc s
                x + y
                q
            s
        r
    p
```



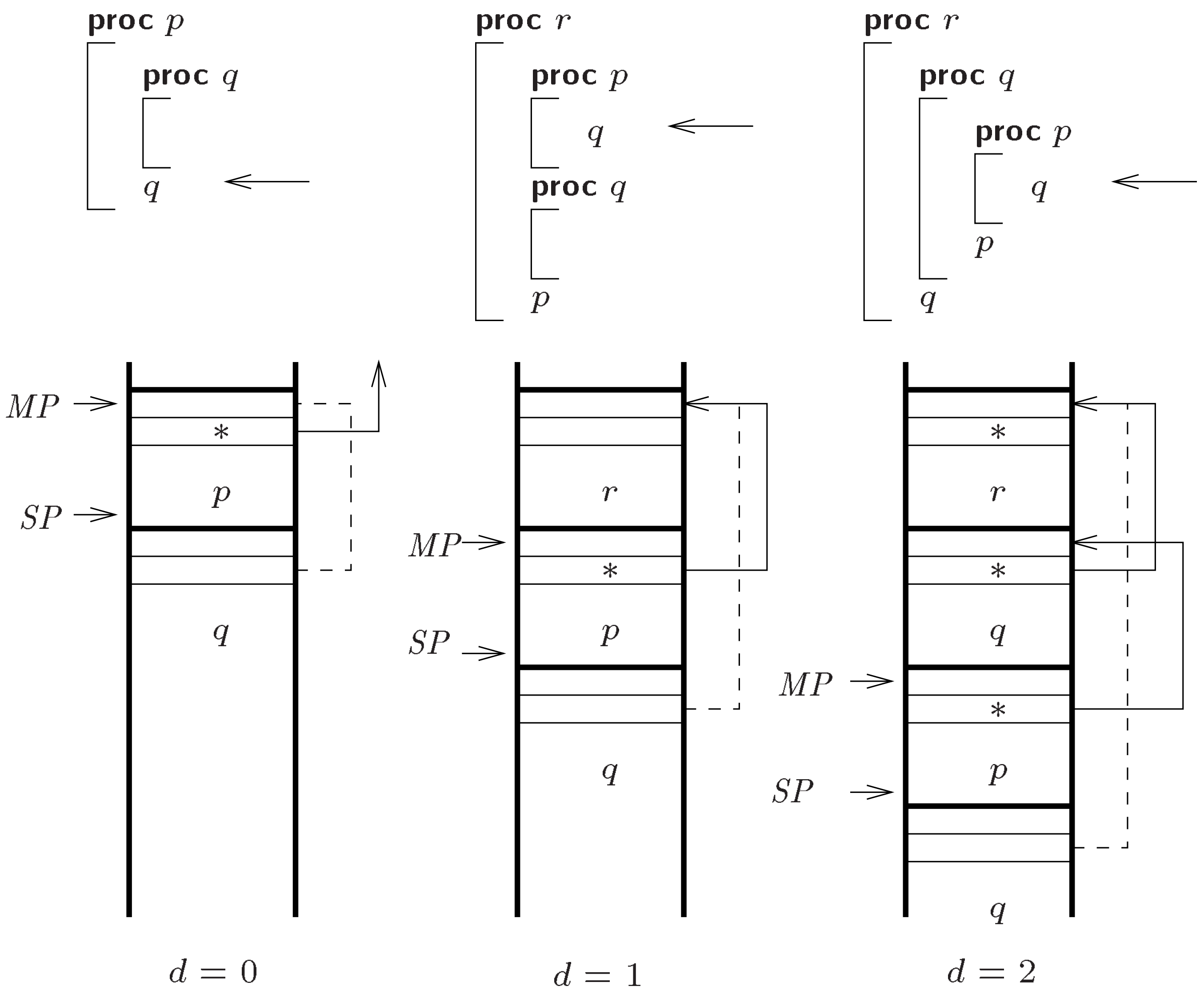


program | stack configuration | corresp. tree of static pred.

Defining	occurrence	Applied	occurrence
p	1	p	1
q	2	q	4
r	2	r (in p)	2
s	3	r (in q)	3

Nesting depths

Updating the Static Link



Three different calls and stack configurations.

Adapt Memory Allocation Scheme

- $\rho(x) = \langle relative\text{-}address, nesting\ depth \rangle$
- Setting relative address remains:

$$\rho_1(v_i) = 5 + \sum_{j=0}^{i-1} size(t_j)$$

for **var** $v_0 : t_0; v_1 : t_2; \ldots; v_k : t_k;$

Size Setting

Static sizes:

Parameters

var-parameters $size = 1$
value-dyn-array-parameters $size = 3 * n + 2$
other-value-parameters $size$ is unchanged

Locals

static-array $size = 3 * n + 2 +$ old-size
other-local $size$ is unchanged

$3 * n + 2$ size of array descriptor

Dynamic sizes:

Dynamic arrays (locals+parameters)
as given by old $size$ formula, but evaluated at run-time.

Adapt $code_L$

Instr.	Meaning
lod $T\ p\ q$	$SP := SP + 1;$ $STORE[SP] := STORE[base(p, MP) + q]$
lda $p\ q$	$SP := SP + 1;$ $STORE[SP] := base(p, MP) + q$
str $T\ p\ q$	$STORE[base(p, MP) + q] := STORE[SP];$ $SP := SP - 1$

$base(p, a)$ = **if** $p = 0$ **then** a **else** $base(p-1, STORE[a+1])$

$code_L(x\ r)\ \rho\ nd =$ **lda** a $d\ ra$;
$code_M\ r\ \rho\ nd$,
where $\rho(x) = (ra, nd')$, $d = nd - nd'$ and
x is variable or formal value parameter

$code_L(x\ r)\ \rho\ nd =$ **lod** a $d\ ra$;
$code_M\ r\ \rho\ nd$
where $\rho(x) = (ra, nd')$, $d = nd - nd'$,
and x is formal var parameter

Code for Procedure/Function Call

1. Set static link of callee
2. Set dynamic link of callee
3. Save EP
4. Evaluate parameters (l-values, r-values)
5. Set MP
6. Save PC as return-address
7. Jump to code of callee
8. Set SP to end of static part
9. Allocate space for dynamic value arrays; copy
10. Set EP

Procedure Call

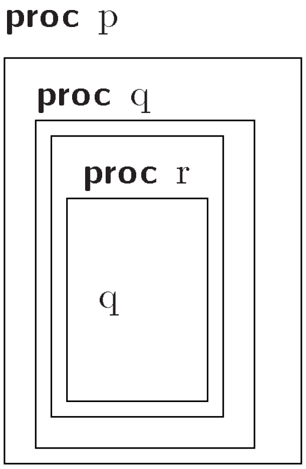
$$
\begin{array}{l}
code\ p(e_1,\ldots,e_k)\ \rho\ nd = \\
\quad \textbf{mst}\ nd - nd'; \\
\quad code_A\ e_1\ \rho\ nd; \\
\qquad \vdots \\
\quad code_A\ e_k\ \rho\ nd; \\
\quad \textbf{cup}\ s\ l \qquad\qquad\qquad (*\ \rho(p) = (l, nd)\ *)
\end{array}
$$

P-instructions for call and entry

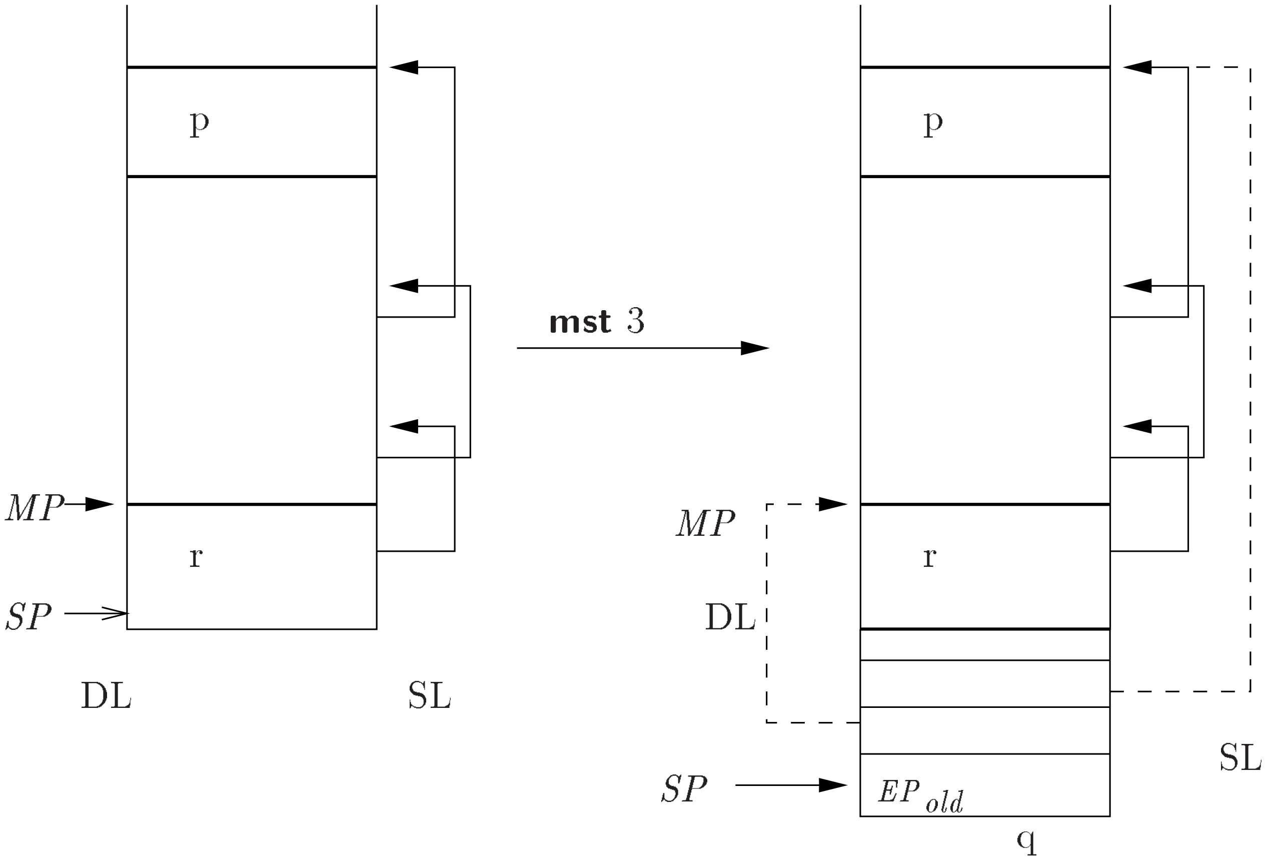
Instr.	Meaning	Comment
mst p	$STORE[SP+2] := base(p, MP);$ $STORE[SP+3] := MP;$ $STORE[SP+4] := EP;$ $SP := SP + 5$	Static link Dynamic link Save EP
cup $p\ q$	$MP := SP - (p+4);$ $STORE[MP+4] := PC;$ $PC := q$	p space for parameters Return address Jump to q
ssp p	$SP := MP + p - 1$	alloc. static area
sep p	$EP := SP + p;$ **if** $EP \geq NP$ **then** *error* (*"store overflow"*) **fi**	alloc. temp. area Check collision stack and heap

$base(p, a)\ =\ \textbf{if}\ p\ =\ 0\ \textbf{then}\ a\ \textbf{else}\ base(p-1, STORE[a+1])\textbf{fi}$

P-instruction mst



nd(applied occurrence of q) -
nd(defining occurrence of q) = 3



Code for Procedure/Function Return

1. Restore SP to the beginning of current stack frame
2. Restore PC to return-address
3. Restore EP and check for heap-stack collision
4. Release frame, i.e. set MP to dynamic link

P-instructions for Return

Instr.	Meaning	Comment
retf	$SP := MP$;	result is on top
	$PC := STORE[MP+4]$;	return address
	$EP := STORE[MP+3]$;	Restore EP
	if $EP \geq NP$	
	then *error*(*"store overflow"*)	
	fi	
	$MP := STORE[MP+2]$	Release frame
retp	$SP := MP - 1$;	No return value
	$PC := STORE[MP+4]$;	return address
	$EP := STORE[MP+3]$;	Restore EP
	if $EP \geq NP$	
	then *error*(*"store overflow"*)	
	fi	
	$MP := STORE[MP+2]$	Release frame

Procedure/Function Code

$$
\begin{array}{l}
code\ (\textbf{procedure}\ p\ (specs);\ vdecls;\ pdecls;\ body)\ \rho\ nd = \\
\qquad \textbf{ssp}\ n\_a'';\\
\qquad code_P\ specs\ \rho'\ nd;\\
\qquad code_P\ vdecls\ \rho''\ nd;\\
\qquad \textbf{sep}\ \ k;\\
\qquad \textbf{ujp}\ \ l;\\
\qquad proc\_code; \qquad\qquad\qquad\qquad (*\ local\ procedures\ *)\\
\quad l: code\ body\ \rho'''\ nd;\\
\qquad \textbf{retp/retf}\\
\quad \text{where} \qquad (\rho', n\_a') = elab\_specs\ specs\ \rho\ 5\ nd\\
\qquad\qquad\qquad (\rho'', n\_a'') = elab\_vdecls\ vdecls\ \rho'\ n\_a'\ nd\\
\qquad\qquad\qquad (\rho''',\ proc\_code) = elab\_pdecls\ pdecls\ \rho''\ nd
\end{array}
$$

$elab\_specs$

$$elab\_specs : \; Spec^* \times \; Addr\_Env \; \times \; Addr \; \times \; ND \; \rightarrow$$
$$Addr\_Env \; \times \; Addr$$

$$elab\_specs \; (\mathbf{var} \; x : t; \; specs) \; \rho \; n\_a \; nd =$$
$$elab\_specs \; specs \; \rho[\,(n\_a, nd)/x\,] \; (n\_a + 1) \; nd$$

$$elab\_specs \; (\textsf{\textbf{value}} \; x\textsf{\textbf{:}} \; \textsf{\textbf{array}}[l_1..u_1, \ldots, l_k..u_k] \; \textsf{\textbf{of}} \; t'; \quad specs)$$
$$\rho \; n\_a \; nd =$$
$$elab\_specs \; specs \; \rho' \; (n\_a + 3k + 2) \; nd \; where$$
$$\rho' = \rho[(n\_a, nd)/x]$$
$$[(n\_a + 2i + 1, nd)/l_i]_{i=1}^{k}$$
$$[(n\_a + 2i + 2, nd)/u_i]_{i=1}^{k}$$

$$elab\_specs \; (\textsf{\textbf{value}} \; x : t; \; specs) \; \rho \; n\_a \; nd =$$
$$elab\_specs \; specs \; \rho[\,(n\_a, nd)/x\,] \; (n\_a + size(t)) \; nd$$
$$\text{for static type } t$$

$$elab\_specs \; (\;) \; \rho \; n\_a \; nd = (\rho, n\_a)$$

$elab\_pdecls$

$elab\_pdecls$ processes procedure declarations

$$elab\_pdecls : Pdecl^{*} \times \; Addr\_Env \; \times \; ND \rightarrow \\ \quad Addr\_Env \; \times \; Code$$

$$\begin{array}{l} elab\_pdecls \; (\; \textbf{proc} \;\; p_1(\ldots);\ldots; \\ \qquad\qquad\qquad \vdots \\ \qquad\qquad\quad \textbf{proc} \;\; p_k(\ldots);\ldots; \;) \; \rho \; nd = \end{array}$$

$$\begin{array}{ll} (\rho', & l_1 : \; code \; (\textbf{proc} \; p_1(\ldots);\ldots) \; \rho' \; nd + 1; \\ & \quad \vdots \\ & l_k : \; code \; (\textbf{proc} \; p_k(\ldots);\ldots) \; \rho' \; nd + 1) \end{array}$$

where $\rho' = \rho[(l_1, nd)/p_1, \ldots, (l_k, nd)/p_k]$

$$elab\_pdecls \; (\;) \; \rho \; nd = (\rho, (\;))$$

Parameter Passing

var-actual-parameters

$$code_A\ x\ \rho\ nd =$$
$$\quad code_L\ x\ \rho\ nd$$

value-actual-parameters

$$code_A\ e\ \rho\ nd =$$
$$\quad code_R\ e\ \rho\ nd$$

value-actual-structural-parameters

$$code_A\ x\ \rho\ nd =$$
$$\quad code_L\ x\ \rho\ nd;$$
$$\quad \textbf{movs}\ g$$

P-code for moves

Instr.	Meaning	Cond.	Res.
movs q	**for** $i := q - 1$ **down to** 0 **do** $STORE[SP + i] :=$ $STORE[STORE[SP] + i]$ **od**; $SP := SP + q - 1$	(a)	
movd q	**for** $i := 1$ **to** $STORE[MP + q + 1]$ **do** $STORE[SP + i] :=$ $STORE[STORE[MP + q]$ $+STORE[MP + q + 2] + i - 1]$ **od**; $STORE[MP + q] :=$ $SP + 1 - STORE[MP + q + 2]$ $SP := SP + STORE[MP + q + 1]$		

Copying Dynamic Arrays

$code_P$ (**value** x : **array**$[u_1..o_1, \ldots, u_k..o_k]$ **of** t) $\rho\ nd =$
movd ra;

The Main Program

$$
\begin{array}{rl}
\multicolumn{2}{l}{\mathit{code}\ (\textbf{program}\ p\ (\mathit{specs});\ \mathit{vdecls;\ pdecls;\ body})\ \rho\ 0 =} \\
 & \textbf{ssp}\ n\_a; \\
 & \mathit{code}_P\ \mathit{vdecls}\ \rho\ 1; \\
 & \textbf{sep}\ \ k; \\
 & \textbf{ujp}\ \ l; \\
 & \mathit{proc\_code;} \\
l: & \mathit{code\ body}\ \rho'\ 1; \\
 & \textbf{stp}
\end{array}
$$

$$
\begin{array}{ll}
\text{where} & (\rho, n\_a) = \mathit{elab\_vdecls\ vdecls}\ \emptyset\ 5\ 1 \quad \text{and} \\
 & \quad (\rho',\ \mathit{proc\_code}) = \mathit{elab\_pdecls\ pdecls}\ \rho\ 1
\end{array}
$$

Input Example

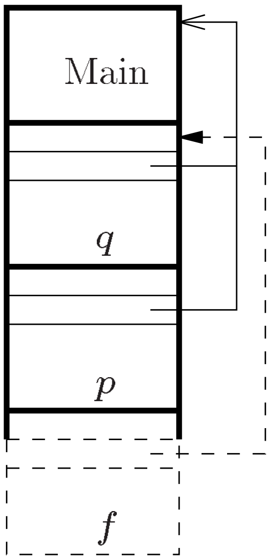
program foo ;
var i : **integer** ;
ssp 10; **sep** 7; **ujp** l_1;
function fact(n: **integer**): **integer**;
l_2: **ssp** 6; **sep** 9; **ujp** l_3;
begin l_3 :
  if $n = 1$
    lda $\mathrm{a}\ 2 - 2\ 5$; **ind** i; **ldc** $i\ 1$; **equ** i; **fjp** l_4
    then $fact := 1$
    lda $\mathrm{a}\ 2 - 2\ 0$; **ldc** $i\ 1$; **sto** i; **ujp** l_5 :
    else $fact := n * fact(n-1)$ **fi**
    l_4: **lda** $\mathrm{a}\ 2 - 2\ 0$; **lda** $\mathrm{a}\ 2 - 2\ 5$; **ind** i;
    mst $2 - 1$; **lda** $\mathrm{a}\ 2 - 2\ 5$; **ind** i; **ldc** $i\ 1$; **sub** i; **cup** $1\ l_2$
    mul i; **sto** i; l_5:
end; **retf**
begin l_1 :
  $i := fact(2)$
  lda $\mathrm{a}\ 1 - 1\ 9$;
  mst $1 - 1$; **ldc** $i\ 2$; **cup** $1\ l_2$
  sto i
end. **stp**

Procedures/Functions as Parameters

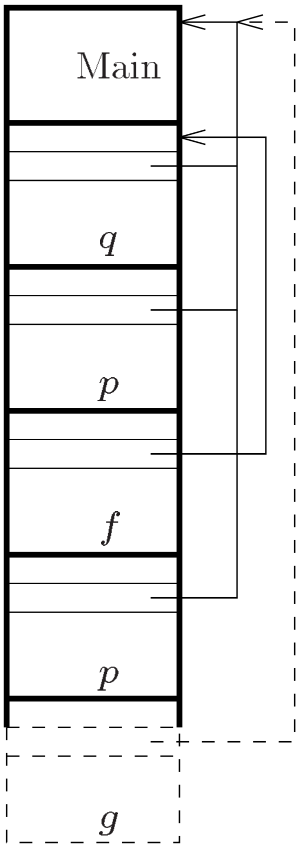
```
program      Main;
    proc p( function h)
        h
    proc q
        function f
            p(g)
        p(f)
    function g

    q
```

Program with
formal procedures



Stacksituation after call $p(f)$ and (dashed) after call h



Stacksituation after call $p(g)$ and (dashed) after call h