Lecture 15: Run-Time Stack

CSC 131

Kim Bruce

Parameter Passing

- Call-by-reference (FORTRAN, Pascal, C++)
  - pass address (l-value) of parameter
- Call-by-copying (Algol 60, Pascal, C, C++)
  - call-by-value/result/ or value-result
  - pass (r-)value of parameter
  - options: in, out, in-out
- Call-by-name (Algol 60)
  - pass actual expression (as "thunk") - not macro!
  - re-evaluate at each access
  - lazy gives efficient implementation if no side effects

Call-by-name

```
procedure swap(a, b : integer);
  var temp : integer;
  begin
   temp := a;
   a := b;
   b := temp
  end;
```

- Won't always work!
- swap(i, z[i]) with i = 1, z[1] = 3, z[3] = 17
- Can't write swap that always works!

What about Java?

- Conceptually call-by-sharing
- Implemented as call-by-value of a reference

Static Memory allocation

- FORTRAN
  - All storage known at translation time
  - Activation records directly associated with code segments
  - At compile time, instructions and vbles accessed by (unit name, offset)
  - At link time, resolve to absolute addresses.
  - Procedure call and return straightforward

Stack-based Allocation

- Pascal, C, C++, Java, ...
  - Activation records on stack
  - Problem: static (scope) vs dynamic (return address)
  - Activation records pushed on call and popped on return
  - Activation record contains:
    - return address
    - return-result address -- *if necessary - where to find result*
    - control or dynamic link -- *to next stack frame*
    - access or static link -- *to nearest stack frame of enclosing scope*
    - parameters, local vbles, & intermediate results.

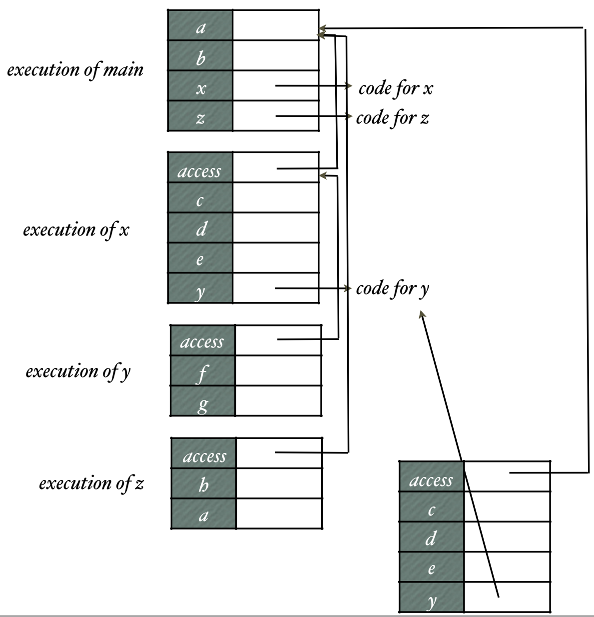
Accessing non-local vbles

```
program main;
   type array_type = array [1..10] of real;
   var a : integer;
        b : array_type;
   procedure x (var c : integer; d : array_type);
   |  var e : array_type;
   |  procedure y (f : array_type);
   |  |  var g : integer;
   |  |  begin
   |  |     z(a+c);
   |  |  end; {y}
   |  begin {x}
   |     : ..... := b[6].......
   |     y(e);
   |  end; {x}
   procedure z (h : integer);
   |  var a : array_type;
   |  begin
   |        :
   |     x (h,a);
   |        :
   |  end;
   begin {main}
         :
      x (a,b);
         :
   end. {main}
```

Assign Variables Offsets

Name	Level	Name	Level
main	0	y	2
a	1	f	3
b	1	g	3
x	1	z	1
c	2	h	2
d	2	a	2
e	2		

Look at run time stack when: main calls x, which calls y, which calls z, which calls x, which calls y, ...



Accessing non-local Variables

- Length of access chain from any fixed procedure to main is always same length
- Any non-local variable found after some fixed number of access links (independent of activation record)
- # of links = constant determinable at compile time
- Access via <chain position, offset> where 1st is # of access links to traverse, 2nd is offset in activation record.

Allocating Activation Record

- Static - sizes of all local variables and parameters known at compile time
  - Fixed size activation records
- Size known at unit activation
  - Array bounds depend on parameters
  - Space for parameter descriptors at fixed offset
- Dynamic
  - Flexible arrays and pointers
  - Allocate on heap w/reference on stack

Tail-Recursive Functions

- Recursion less efficient (space & time) than iteration because of activation records.
- A call to function f in body of g is tail if g returns immediately after call of f terminates.
  - Ex: fun g x = if x > 0 then f x else f (-x)
- Tail calls use stack space more efficiently
- Tail recursive functions even better!

Tail-recursive Functions

- *Compare*:

```
rev [] = []
rev (fst:rest) = (rev rest)++[fst]
```

- *and*

```
reverse l = tlrev l [] where
              tlrev [] r = r
              tlrev (fst:rest) r =
                     tlrev rest (fst:r)
```

- *Can accumulate answer ...*

```
fact n = tlfact(n,1)
tlfact (n,ans) = if n <= 1 then ans
                      else tlfact(n-1,n*ans);
```

Fibonacci

```
int fib(int n) {
  int current = 1;
  int next = 1;
  while (n > 0) {
    int temp = current;
    current = next;
    next = next + temp;
    n = n - 1;
  }
  return current;
}
```

or recursively:

```
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib(n-2);
```

Replace while loops

Can replace while by tail recursive function where all variables used become parameters:

```
fastfib n = fibloop n 1 1
```

where

```
fibloop 0 current next = current
fibloop n current next =
              fibloop (n-1) next (current + next);
```

Correctness

- Let $a_0, a_1, \ldots$ be list of Fibonacci numbers
- Lemma: For all $n, k \geq 0$,

  $\text{fibloop } n\ a_k\ a_{k+1} = a_{k+n}$

- fastfib n = fibloop n 1 1

  = fibloop n a_0 a_1

  = a_n

Function Parameters

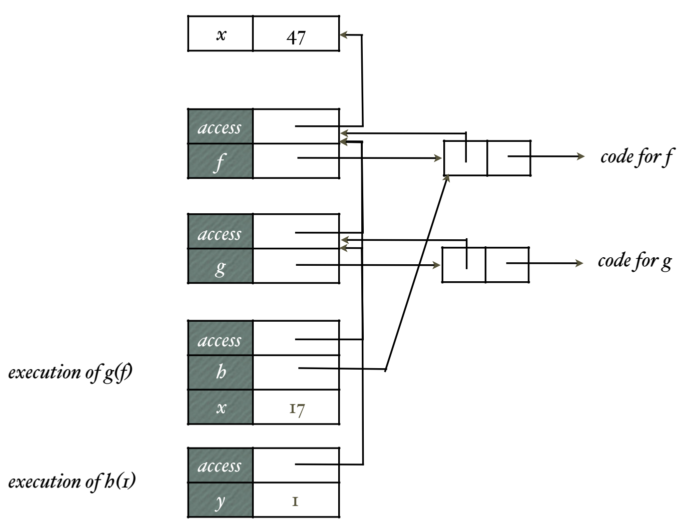
- Harder to cope with because need environment defined in. Two problems:
  - Downward funarg:

```
x = 47
f y = x + y;
g(h) = let val x = 17
       in h(1)
> g(f)
```

  - When evaluate f(1), is in environment where x = 17!
- Return function value -- loses env of definition

Represent function values as closures

- Function value represented as a pair of
  - Environment (pointer to run-time stack where defined)
  - Code for function
- When call a function (passed as closure)
  - Allocated activation record for function
  - Set access link in activation record using value in closure.



Function as Return Value

```
fun make_counter(init: int) =
    let
        val count = ref init
        fun counter(inc:int) =
                (count := !count + inc; !count)
    in
        counter
    end;
```

ML program

```
val c = make_counter(1);
c(2) + c(2);
```

c needs access to count when applied!
Stack discipline does not work.

```
                 ----------------
                 v               |
          ----------          --|--------
 make_ctr |    ----|--------- | |  |  --|------> code for make_ctr
          ---------- <-       ----------
                      |
          ----------  |
   access |    ----|--|
          ----------<--|      ----------
        c |    ----|---|----- | |  |  --|------> code for counter
          ----------   |      --|--------
                       |        | ^
mk_ctr(1) ----------   |        | |
   access |     ----|----       | |
          ---------- <----------  |
     init |   1     |             |
          ----------     -------  |
    count |     ----|--  |  1  |  |
          ----------     -------  |
  counter |     ----|--------------|
          ----------
```

While executing next to last line of program: c = mk_ctr(1)
Just before assign to c

```
                 ----------------
                 v               |
          ----------          --|--------
 make_ctr |    ----|--------- | |  |  --|------> code for make_ctr
          ---------- <-       ----------
                      |
          ----------  |
   access |    ----|--|
          ----------<--|      ----------
        c |    ----|---|----- | |  |  --|------> code for counter
          ----------   |      --|--------
                       |        | ^
mk_ctr(1) ----------   |        | |
   access |     ----|----       | |
          ---------- <----------  |
     init |   1     |             |
          ----------     -------  |
    count |     ----|--  |  1  |  |
          ----------     -------  |
  counter |     ----|--------------|
          ----------
```

When make assignment c = mk_ctr(1),
pop off activation record for mk_ctr(1) ...

```
                 ----------------
                 v               |
          ----------          --|--------
 make_ctr |    ----|--------- | |  |  --|------> code for make_ctr
          ---------- <-       ----------
                      |
          ----------  |
   access |    ----|--|
          ----------<--|      ----------
        c |    ----|---|----- | |  |  --|------> code for counter
          ----------   |      --|--------
                       |        | ^
mk_ctr(1) ----------   |        | |
   access |     ----|---^       | |
          ---------- <--| -------  |
     init |   1     |   |         |
          ----------    | -------  |
    count |     ----|-- ||  1  |  |
          ----------    | -------  |
  counter |     ----|---|----------|
          ----------    |
                        |
c(2)      ----------    |
   access |     ----|---
          ----------
      inc |   2     |
          ----------
```

Still needed, but no longer there!
Must be retained even though not on run-time stack!!!!

Problem

- When call c(2), activation record for make_counter is gone.
- Hence no access to count
- To solve, must keep activation records around for functions that return functions
- Garbage collect them when no longer reference to them!