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

```plaintext
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}
  : a(1), b[6], c(2), d(2), e(2)
end {main}
```

Assign Variables Offsets

<table>
<thead>
<tr>
<th>Name</th>
<th>Level</th>
<th>Name</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
<td>0</td>
<td>y</td>
<td>2</td>
</tr>
<tr>
<td>a</td>
<td>1</td>
<td>f</td>
<td>3</td>
</tr>
<tr>
<td>b</td>
<td>1</td>
<td>g</td>
<td>3</td>
</tr>
<tr>
<td>x</td>
<td>1</td>
<td>z</td>
<td>1</td>
</tr>
<tr>
<td>c</td>
<td>2</td>
<td>h</td>
<td>2</td>
</tr>
<tr>
<td>d</td>
<td>2</td>
<td>a</td>
<td>2</td>
</tr>
<tr>
<td>e</td>
<td>2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*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:
  \[
  \text{rev} \; \text{[]} = \text{[]} \\
  \text{rev} \; (\text{fst:rest}) = (\text{rev } \text{rest})++[\text{fst}]
  \]
- and
  \[
  \text{reverse } \text{l} = \text{tlrev } \text{l } \text{[]} \text{ where} \\
  \text{tlrev } \text{[]} \; r = r \\
  \text{tlrev} \; (\text{fst:rest}) \; r = \text{tlrev } \text{rest} \; (\text{fst:r})
  \]
- Can accumulate answer ...
  \[
  \text{fact} \; n = \text{tlfact}(n,1) \\
  \text{tlfact} \; (n,\text{ans}) = \text{if } n <= 1 \text{ then } \text{ans} \\
  \quad \text{else } \text{tlfact}(n-1,n*\text{ans});
  \]

Replace while loops

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

- Let \(a_0, a_1, \ldots\) be list of Fibonacci numbers
- Lemma: For all \(n, k \geq 0\),
  \[
  \text{fibloop } n \; a_k \; a_{k+1} = a_{k+n}
  \]
- \[
  \text{fastfib } n = \text{fibloop } n \; 1 \; 1 \\
  \begin{align*}
  \text{fibloop } 0 \; \text{current} \; \text{next} &= \text{current} \\
  \text{fibloop } n \; \text{current} \; \text{next} &= \text{fibloop } (n-1) \; \text{next} \; (\text{current} + \text{next});
  \end{align*}
\]

Correctness

Fibonacci

\[
\begin{align*}
\text{int fib}(\text{int } n) \{ \\
\text{int current} &= 1; \\
\text{int next} &= 1; \\
\text{while } (n > 0) \{ \\
&\text{int temp} = \text{current}; \\
&\text{current} = \text{next}; \\
&\text{next} = \text{next} + \text{temp}; \\
&n = n - 1;
\} \\
\text{return current;}
\}
\end{align*}
\]

or recursively:

\[
\begin{align*}
\text{fib } 0 &= 1 \\
\text{fib } 1 &= 1 \\
\text{fib } n &= \text{fib } (n-1) + \text{fib}(n-2);
\end{align*}
\]
**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;

val c = make_counter(1);
val c2 = c(2) + c(2);  
```

ML program

```
c needs access to count when applied!
Stack discipline does not work.
```
While executing next to last line of program: c = mk_ctr(1)
Just before assign to c

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!