Lecture 15: Run-Time Stack

CSC 131

Kim Bruce

Midterm

- Open book, notes, course web pages
- The exam will be available by 9 a.m. Monday and must be turned in electronically by midnight Thursday.

Midterm Topics

- Haskell (including monads/type-classes)
- Language implementation
  - lexing/parsing/type-checking & inference/interpreters
- Lambda calculus
- Run-time memory management
  - Run-time stack (no function arguments/results)
  - heap on final

Function Parameters

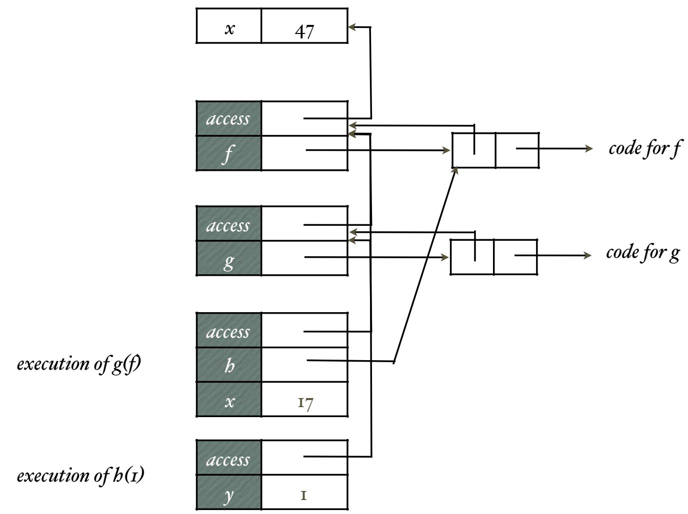
- Harder to cope with because need environment defined in. Two problems:
  - Downward funarg:

```
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);
c(2) + c(2);
```

ML program

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!

Dynamic Languages

- Dynamic scope -- no longer need static/access link in activation record
  - look for closest activation record with vble
  - must be able to find names dynamically
- Dynamic types -- associate type descriptor w/ values of variables
- Late binding costs -- more space, slower access
- Benefits - more flexibility

Heap Management

- Stack doesn't work in some circumstances
  - functions returning functions
  - dynamically allocated memory
- Heap allows dynamic allocation/deallocation of memory.
  - Manually
  - Automatically

Managing the Heap

- Heap maintained as stack of blocks of memory
- Need strategy to handle requests and returns.
  - Best fit
  - First fit
- Fragmentation is serious problem when return
- Coalesce blocks on heap
- May need to compact memory occasionally

Automating Dispose

- Garbage collection (lazy)
  - LISP by McCarthy
- Reference counting (eager):
  - Keep track of number of refs to block of memory.
  - Return it when count is 0.
  - Disadvantages:
    - space and time overhead of keeping count,
    - circular structures.
  - Weak variant used in Objective C on iphone
    - Newest version automates it.
    - Python uses ref counting + GC for circular

Garbage Collection

- At a given point in execution of program P, memory location m is garbage if no continued execution of P from this point can access m.
- Automatic garbage collectors start with root set and search out all memory locations accessible from root set.
- Automatic garbage collectors necessarily conservative.

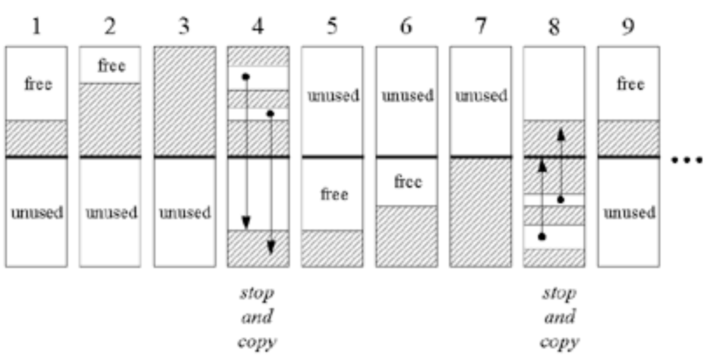
Mark and Sweep Collector

- Mark "alive" elements.
- Sweep through memory and reclaim garbage
- Problems:
  - Space for marks (and stack while marking)
  - Two sweeps through memory needed
  - Sweeping takes time proportional memory size
- Used in Java 1.0, 1.1, but not later

Copying Collector

- Divide memory in half -- working vs. free
- When working exhausted
  - Copy live nodes from working to free (use forwarding address)
  - Swap halves
- Evaluation:
  - Only looks at live cells, but can be incremental
  - Needs twice as much space, but respects cache
  - Allocation very cheap! Always one big block free
  - GC fast if most are dead

Copying Collector



Memory as time passes ...

Diagram from Bill Venners, Inside the Java VM

Generational Collector

- Only try to collect recently allocated blocks
  - Infant mortality - majority of blocks die young!
- Divide memory into two or more generations.
- Modern Java uses copying collector for youngest and older uses mark-compact scheme
  - youngest gets lots of garbage quickly
  - mark-compact doesn't move lots of older objects
  - Can now hand-tune GC

Implementing Parametric Polymorphism

Section 6.4.2 of text

Parametric Polymorphism Redux

- How do we implement polymorphic classes, functions, etc.
- Scheme, ML, Haskell, Clu (1974), Ada, C++, Eiffel, Java
- Efficient implementation depends on shared code.

C++ templates

```
template <typename T>
class Stack {
  private:
    std::vector<T> elems;     // elements

  public:
    void push(T const&);      // push element
    void pop();               // pop element
    T top() const;            // return top element
    bool empty() const {      // return if stack empty
        return elems.empty();
    }
};
```

Different T's take different amounts of space, so macro-expand at compile time

Easier if Uniform Reps

- LISP, Scheme, ML, Haskell, Clu, Eiffel, and Java have uniform reps for values so can share same code.
- Ada requires different implementation, but still type-checks statically.
- Automatic boxing and unboxing helps with primitives.