

Lecture 23: More Higher-Order Functions & Managing the Heap

CSCI 131
Spring, 2011

Kim Bruce

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;

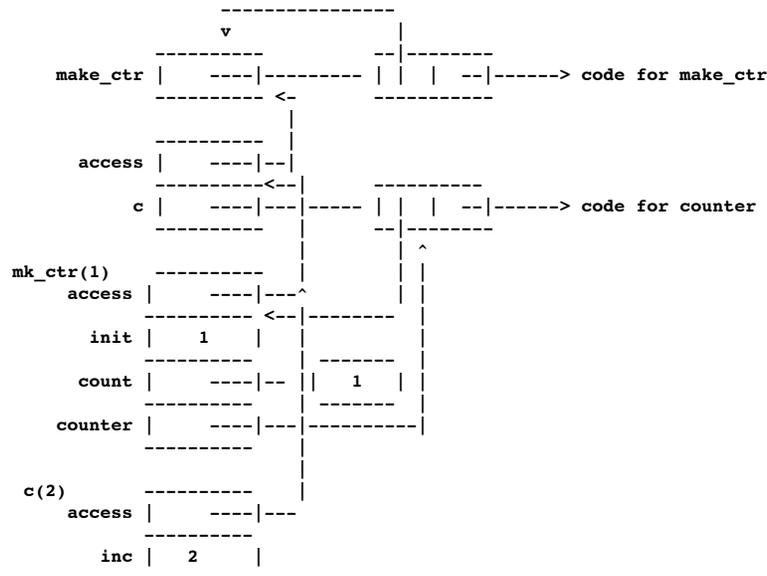
val c = make_counter(1);
c(2) + c(2);
```

*ML
program.*

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

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 done.



Dynamic Languages

- Dynamic types -- associate type descriptor w/ values of variables
- Dynamic scope -- no longer need static/access link in activation record
 - look for closest activation record with vble
- 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)
- Reference counting (eager):
 - Keep track of number of references 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

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.

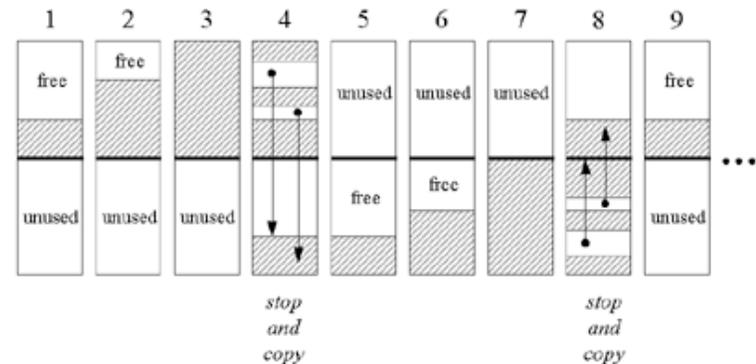
Mark and Sweep Collector

- Mark “alive” elements.
- Sweep through memory and reclaim garbage
- Problems:
 - Space for marks (and stack while marking)
 - Two sweeps through memory needed
 - Time proportional to # of live nodes
- 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 ...

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

Implementing Parametric Polymorphism

Section 6.4 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.