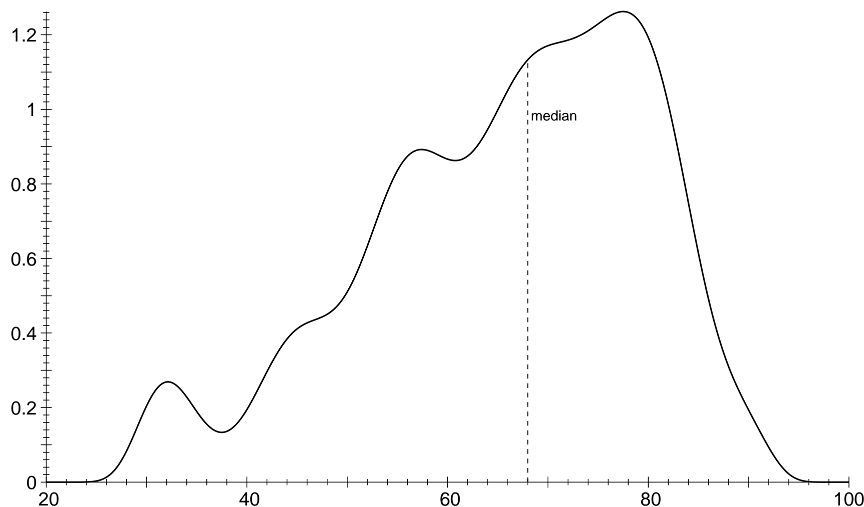
Preliminary Examination II
Computer Science 312, Cornell University 17 April 2007

Solutions

Overall, student performance was mixed, but generally lower than the first prelim. The smoothed histogram of scores looked like this (vertical axis is number of students per unit score):



1. Cookie [1 pt]

   What kind of cookie did you take?

   Answer:

   We accepted any answer except "none"

2. True/False [12 pts]

   (parts a–f; 2 points off for each wrong answer, 1 point off for each blank answer)

   (a) A hash table can efficiently implement an immutable set abstraction.

   False

   (b) If using reference counting, an imperative update to a pointer variable causes updates to two reference counts.

   True

   (c) Balanced binary tree data structures ensure that every leaf is at the same depth from the root.

   False

   (d) With a balanced binary tree, it is possible to find all elements in the tree in order in time linear in the size of the tree.

True – in fact, this is true of any binary search tree

(e) If a function is $O(4^n)$, it is also $O(2^n)$.

False

(f) If a function is $O(n)$, it cannot be $\Omega(n^2)$.

True

3. Environment model [30 pts] (parts a–c)

Consider the following SML program:

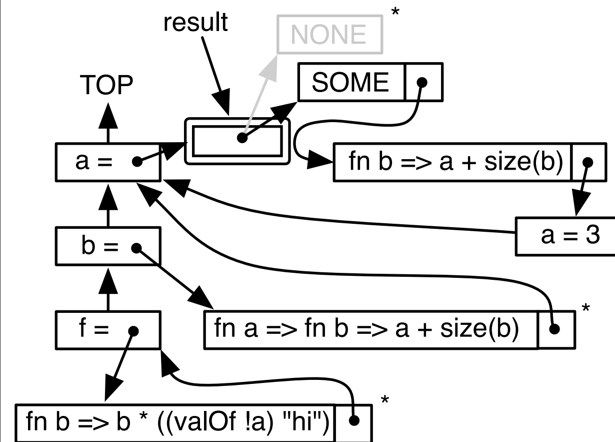
```
let val a: (string->int) option ref = ref NONE
    val b: int->string->int = fn a:int => fn b:string => a + String.size(b)
    fun f(b: int):int = b * ((valOf !a) "hi")
in
    a := SOME(b(3));
    a
end
```

(a) [5 pts] Circle each bound variable occurrence in the above code and draw an arrow pointing to its binding occurrence.

(b) [20 pts] Draw the result produced by evaluating this expression in the environment model, by completing the following diagram. Show all boxes created during evaluation, and draw an arrow from the word "result" to the result value or box.

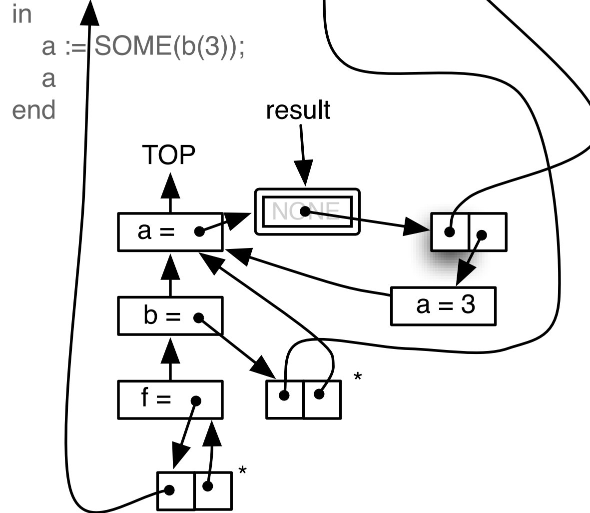
Answer:

Here is the way to draw the diagram shown in class:



A clever implementation of SML like SML/NJ can actually avoid creating boxes for a datatype like `option` *in which there is only one non-unit constructor. The figure is also a little misleading because it suggests that code is copied as functions are evaluated. In fact, the following diagram is a more accurate depiction of how closures and options are represented. Closures just contain a pointer to program code that is fixed*

```
let val a: (string->int) option ref = ref NONE
   val b: int->string->int = fn a:int => fn b:string => a + String.size(b)
   fun f(b: int):int = b * ((valOf !a) "hi")
in
   a := SOME(b(3));
   a
end
```



and unchanging.

(c) [5 pts] Draw an asterisk (*) next to any box in the diagram that is allocated on the heap and becomes garbage once the program is done (assume the result is a root).

Answer:

See the diagrams above.

4. Mutable data abstractions [24 pts] (parts a–d)

Suppose we want to implement a mutable set abstraction, with the following signature:

```
signature MUTABLE_SET = sig
  (* An 'a mset is a mutable set of elements of type 'a *)
  type 'a mset

  (* create(eq) creates an empty mutable set in which elements are
   * compared for equality using eq. *)
  val create : ('a * 'a -> bool) -> 'a mset

  (* add(s, x) adds the element x to s. Returns true if s already
   * contained x, false otherwise. *)
  val add: 'a mset * 'a -> bool

  (* contains(s, x) returns whether s contains x.  *)
  val contains: 'a mset * 'a -> bool
end
```

(a) [5 pts] Given the data structures we have seen, what is the best average-case asymptotic performance (amortized or not) we can expect for the `add` and `contains` operations? Explain your answer briefly.

Answer:
Both operations need to determine if x is in the set. Because there is nothing we can do to elements except test their equality (we don't have ordering or hash codes!), we can't check for the presence of an element any more effectively than by testing it against every element in the set. Therefore, we can't implement either operation in better than $O(n)$ time. A linked list will achieve this asymptotic performance.

(b) [5 pts] How might we change the `create` operation to allow a faster implementation? Write a new type and specification for `create` that makes this possible, and give the improved asympotic performance of the other operations.

Answer:
We could implement this abstraction with a hash table if create were provided a hashcode operation:

```
(*  create(eq, hashcode) creates a new mutable set of elements of
 *  type 'a. The function eq implements an equality test on elements,
 *  and hashcode returns a hashcode. Requires: equal elements have the
 *  same hashcode. *)
val create: ('a * 'a -> bool) * ('a -> int) -> 'a mset
```

Suppose we want to implement `MUTABLE_SET` but we already have a good *immutable* set implementation named `Set` handy. This implementation and its signature (`SET`) are defined as follows:

```
structure Set :> SET = struct
  ... (* you don't need to know *) ...
end
signature SET = sig
  (* An 'a set is an immutable set of elements of type 'a *)
  type 'a set
   (* create(eq) creates an empty set in which elements are
   * compared for equality using eq. *)
  val create : ('a * 'a -> bool) -> 'a set
   (* add(s, x) returns (s', f) where s' contains the elements of s
   * and also x. Returns true if s already contained x, false if it did not. *)
  val add: 'a set * 'a -> bool
   (* contains(s, x) returns whether s contains x.  *)
  val contains: 'a set * 'a -> bool
end
```

(c) [10 pts] Show how to implement `MUTABLE_SET` simply, using `Set` and values of the type `Set.set`. Include the abstraction function and rep invariant.

Answer:

```
structure MutableSet :> MUTABLE_SET = struct
  type 'a mset = 'a Set.set ref
  (* AF: an mset contains the elements in the set it refers to.
   * RI: none *)
  fun create(eq) = ref (Set.create(eq))
  fun add(s, x) =
    let val (s', b) = Set.add(!s, x) in
        s := s';
        b
    end
  fun contains(s, x) = Set.contains(!s, x)
end
```

(d) [4 pts] Suppose that the `MUTABLE_SET` signature also included the following operation:

```
(* copy(s) returns a new set containing the same elements as s.
 * Performance: O(1) time.
 *)
val copy: 'a mset -> 'a mset
```

Show how to extend your `MutableSet` implementation with an implementation of this constant-time operation.

Answer:

```
fun copy(s) = ref(!s)
```

5. Asymptotic complexity [20 pts] (parts a–b)

(a) [15 pts] Consider this recurrence:

$$T(0) = T(1) = 1$$
$$T(n) = \lg n + T(n/2)$$

Show that $T(n)$ is $O(\lg^2 n)$. (Recall $\lg^2 x = (\lg x)^2$.) As in the problem set, you may assume that n is a power of two (i.e., $n/2$ is an integer).

Answer:

We use the substitution method, observing that $\lg^2 n$ is monotonic in n. We need to show that for sufficiently large n, there exists a k such such that $k\lg^2 n \geq \lg n + k\lg^2(\frac{n}{2})$. This can be manipulated as follows:

$$\begin{aligned}
k\lg^2 n &\geq \lg n + k\lg^2(\frac{n}{2}) \\
k\lg^2 n &\geq \lg n + k(\lg n - 1)^2 \\
k\lg^2 n &\geq \lg n + k\lg^2 n + k - 2k\lg n \\
0 &\geq \lg n + k - 2k\lg n \\
0 &\geq \lg n(1-2k) + k
\end{aligned}$$

Now, if we choose $k > 1/2$, the $\lg n$ will be negative and will dominate the k term for large n. Therefore $T(n)$ is $O(\lg^2 n)$.

(b) [5 pts] Show that $T(n)$ is also $\Theta(\lg^2 n)$.

Answer:
If we use the same argument (except with $\leq$ instead of $\geq$), and choose $k < 1/2$, then the $\lg n$ term is positive. Therefore $T(n)$ is $\Omega(\lg^2 n)$, and since it's bounded above and below, it's $\Theta(\lg^2 n)$.

6. Memory layout [13 pts]

Suppose that you are implementing the run-time system for a programming language similar to SML. You have available n 32-bit words of memory with addresses ranging from 0 to $4(n-1)$. Each address corresponds to a real memory location (the system does not have virtual memory). The stack grows downward from the top of the address space, and the heap grows upward from some address h_0. Suppose that the minimum size of a heap memory block is 4 words, and the heap and the stack take up h and s words respectively. What is the minimum amount of unused space between the heap and the stack at which you can run a mark-and-sweep garbage collection and be sure that you will succeed? Explain and justify any assumptions you make.

Answer:
We assume a simple mark-and-sweep that does not do pointer reversal or do tail-recursion on the last pointer out of an object. Then the worst case is when the whole heap consists of one big linked list. Assume that the linked list nodes take up 4 words of memory. Then there are at most $h/4$ list nodes. Recursive marking of the linked list will take as many stack frames as there are list elements.

Suppose that a stack frame contains three words (one to keep track of the current object, one to keep track of the index into the object, one for the return address). So we need at least $3h/4$ free space between the stack and the heap to store those stack frames.