Preliminary Examination #2
Computer Science 312, Cornell University 16 April 2002

Solutions

1. Correctness [30 pts]

(a) [5 pts] Make the following specification more clear without ruling out reasonable implementations (such as the implementation that follows.) Do not change the type.

```
(* Finds an element in an array between two positions.
 * Requires: the elements are in increasing order.
 *)
val binSearch : ('a * 'a -> order) * 'a array * 'a * int * int -> int
```

Answer:

```
(* binSearch(compare, a, x, l, r) returns an index i such that
 * Array.sub(a,i) is equal to x according to compare, and l <= i <=
 * r.  If no such i exists, returns some i such that l <= i <= r.
 * Requires: 0 <= l <= r < Array.length(a) , and the elements of a
 * are in increasing order according to compare.
 *
 * Possible alternatives: it's OK to only require
 * 0 <= i < Array.length(a) in the "not found" case; it is also OK
 * to only require the elements in indices l through r (inclusively)
 * to be in increasing order; and the order does not really have
 * to be strictly increasing, just non-decreasing is good enough.
 *)
```

Now, suppose the following is a correct implementation:

```
fun binSearch(compare, a, x, l, r) =
  if l = r
    then r
    else let val m = (l+r) div 2
         in
           case compare(Array.sub(a,m),x) of
             LESS => binSearch(compare, a, x, m+1, r)
           | _ => binSearch(compare, a, x, l, m)
         end
```

(b) [5 pts] Propose and justify 5–8 black-box test cases for unit-testing this function. Choose your test cases carefully so they constitute a reasonable (if small) test suite. Your descriptions and justifications should be very brief; just give an idea of what it is you think each test case is testing.

Answer:

Strategy:

- *test against array of one element, two elements, several elements*
- *test [l,r] bounds encompassing whole array, both in interior, $l = r$*
- *test array containing element, and not containing element*
- *test element contained at beginning of range, end of range, middle of range*
- *test array of elements that are all equal*

Example test cases:

i. `binSearch(Int.compare, [|1,2,3,4,5|], 2, 0, 4)`
ii. `binSearch(Int.compare, [|1,2|], 1, 0, 1)`
iii. `binSearch(Int.compare, [|1,2|], 2, 0, 1)`
iv. `binSearch(Int.compare, [|1|], 1, 0, 0)`
v. `binSearch(Int.compare, [|1,2,3,4,5|], 6, 0, 4)`
vi. `binSearch(Int.compare, [|1,2,3,4,5|], 0, 1, 3)`
vii. `binSearch(Int.compare, [|1,1,1,1,1|], 1, 1, 3)`
viii. `binSearch(Int.compare, [|1,1,1,1,1|], 2, 1, 2)`

Common mistakes: not testing the common-case behavior of the function at all, violating the requires clause (such tests are useless: the function is allowed to do anything it wants then), skipping tests on corner cases for `l` *and* `r` *and array length.*

(c) [10 pts] Show by induction that assuming the precondition is satisfied, a call to this implementation of `binSearch` terminates normally (does not raise an exception) and takes $O(\lg(\texttt{r} - \texttt{l}))$ time. (**Hint:** sure your precondition in part (a) is strong enough!)

Answer:

We show this by induction on `r - l`*, which must be nonnegative by the precondition.*

If `l = r`*, clearly the function terminates in* `then`*. Otherwise, we have* `l>r`*. Because* `l` *and* `r` *are both nonnegative...*

(d) [5 pts] Briefly describe how to use this function to implement an ordered set abstraction with an efficient `member` operation. Provide code to fill in the two boxes below, and give the rep invariant.

```
structure SortedArraySet = struct
  type 'a set = [1]
  fun member((a, compare), e): bool =  [2]
  ...
end
```

Answer:

The representation (1) is just a pair of type `'a array * ('a * 'a -> order)`*. The rep invariant for* `(a,compare)` *is that the array is sorted in ascending order according to* `compare`*.*

The `member` *code (2) can use* `binSearch`*:*

```
fun member((a, compare), e): bool =
  let val i = binSearch(compare, a, e, 0, Array.length(a) - 1)
      val e' = Array.sub(a,i)
  in
      compare(e, e') = EQUAL
  end
```

(e) [5 pts] What is the asymptotic performance of `member`? Briefly describe when this ordered set implementation would be preferable to red-black trees, and vice-versa.

Answer:

`member` *takes* $O(\lg n)$ *time. This set abstraction provides slow, destructive update. So if functional update is needed or update is a time-critical operation, red-black trees are the preferred data structure. This data structure wins if the* `member` *operation is very heavily used, because it uses little memory.*

2. Evaluation models [25 pts]

Consider the following two expressions, where e_1 and e_2 are subexpressions that make both expressions legal SML:

1. `let val z =` e_1 `in (`e_2`)*(`e_2`) end`
2. `let val z =` e_1
   `val y =` e_2
   `in`
   `y*y`
   `end`

(a) [5 pts] Give expressions e_1 and e_2 such that the two expressions give different results.

Answer:

$e_1 =$ `ref 1`

$e_2 =$ `(z := !z + 1; !z)`

(b) [10 pts] Draw environment model diagrams to show why these expressions evaluate differently.

Answer:

(c) [10 pts] Use the substitution model to argue that they always compute the same value! (**Hint:** think about the series of evaluation steps each expression must take.)

Answer:

The first expr evaluates to e_1 to some value v_1, as does the second statement. The first expr then evaluates to $e_2\{v_1/\texttt{z}\} * e_2\{v_1/\texttt{z}\}$. The second expr evaluates $e_2\{v_1/\texttt{z}\}$ to some value v_2 and then computes $v_2 * v_2$. Whatever series of computation steps is taken in the second expr will be taken for each of the multiplicands in the first expr, so the first expr will also evaluate to $v_2 * v_2$.

3. Graph algorithms [26 pts]

(a) [8 pts] It was shown in class that Dijkstra's algorithm works correctly. The argument depended on the replacement of the FIFO queue used in breadth-first search with a priority queue. Suppose that we ran the same algorithm with the original FIFO queue. Draw a small graph for which the algorithm would fail to compute the correct shortest-path distance for some vertex, and explain briefly why.

Answer:

For the rest of the problem, consider the following signature for a directed graph abstraction:

```
signature GRAPH = sig
  type graph
  type vertex
  val vertices: graph->vertex list
  (* outgoing(v) is a list of vertices that are the
   * destination vertices for each edge leaving v.
   * Run time is linear in the length of the result. *)
  val outgoing: vertex->vertex list
  (* identity(v) is the identity of a vertex: two
   * vertices have the same identity iff they are the
   * same vertex. *)
  val identity: vertex->int
   ... other graph, vertex creation operations ...
end
```

(b) [10 pts] Here is a signature for a function that detects cycles in graphs:

```
(* cycleExists(v) is whether there is a path from v to a cycle in the
 * graph of v. A cycle is a sequence vertices (v1, v2, ..., vn) such
 * that there is an edge from each vertex in the sequence to the next,
 * and an edge from vn back to v1. *)
val cycleExists: vertex -> bool
```

Sketch an $O(V+E)$ algorithm for implementing this specification, where V is the number of graph vertices and E is the number of edges. You may assume that standard data structures are available, but indicate which ones you need to use. Keep your answer short.

Answer:

(c) [8 pts] For some graph implementations (e.g., game trees), the list of outgoing vertices may be large and expensive to compute, and yet not needed in its entirety for some algorithms. Suggest an operation to replace `outgoing` in the signature that will provide the same essential functionality, yet avoid the cost of computing vertices that are never used. Give the type and specification of this operation. (There is more than one way to do this.)

Answer:

There are several ways to do this. One is to have `outgoing` *return a stream of vertices (which can be computed lazily):*

```
val outgoing:  vertex -> vertex option stream
```

Another option is to provide a `fold` *operation over the outgoing vertices; the user raises an exception within the folding function if not all the outgoing vertices are to be visited:*

```
val fold_outgoing:  ('b * vertex -> 'b) -> 'b -> vertex -> 'b
```

4. Type checking [19 pts]

For some SML function types, it is tricky to produce an expression with that type—particularly if we require that the expression terminates and cannot throw an exception. For example, given

```
val f : 'a -> 'b -> 'a
```

you could write the following declaration:

```
fun f x y = x
```

If you type that into SML you get the desired type:

```
Standard ML of New Jersey, Version 110.0.7, September 28, 2000
- fun f x y = x;
val f = fn : 'a -> 'b -> 'a
```

However, there is no way to produce a normally terminating SML expression with the type `'a -> 'b`, because it promises to convert a value of any type to a value of any other type.

Below are a number of SML function types. For each one, if you can write a function declaration for which SML will infer the stated type, do so. If not, state clearly that it is impossible. Your functions must terminate and cannot throw exceptions. Recall that `->` is right-associative.

(a) [3 pts] `val f : 'a -> 'a`

Answer:

`fun f x = x` *or* `val f = fn(x)=>x`

(b) [4 pts] `val f : 'a -> 'b -> 'b`

Answer:

`fun f x y = y` *or* `val f = fn(x)=>fn(y)=>y`

(c) [4 pts] `val f : ('a -> 'b) -> 'b -> 'a`

Answer:

impossible

(d) [4 pts] `val f : ('a -> 'b) -> 'a -> 'b`

Answer:

`fun f g x = g x` *or* `val f = fn(g)=>fn(x)=>g x`

(e) [4 pts] `val f : ('a -> 'b) -> ('b -> 'c) -> ('a -> 'c)`

Answer:

`fun f g h x = h(g x)` *or* `val f = fn(f) => fn(g) => fn(x) => g(f x)` *or* `val f g h = h o g`