Recitation 11: Mutable Data Structures

Thus far, we've been working with the purely functional fragment of SML. That is, we've been working with the subset of the language that does not include computational effects (also known as side effects), ignoring for the moment print and infinite loops. In particular, whenever we coded a function, we never changed variables or data. Rather, we always computed new data. For instance, when we wrote code for an abstract data type such as a stack, queue, or dictionary, the operations to insert an item into the data structure didn't effect the old copy of the data structure. Instead, we always built a new data structure with the item appropriately inserted. (Note that the new data structure might refer to the old data structure, so this isn't as inefficient as it first sounds.)

For the most part, coding in a functional style (i.e., without side effects) is a "good thing" because it's easier to reason locally about the behavior of the code. For instance, when we code purely functional queues or stacks, we don't have to worry about a non-local change to a queue or stack. However, in some situations, it is more efficient or clearer to destructively modify a data structure than to build a new version. In these situations, we need some form of mutable data structures.

Like most imperative programming languages, SML provides support for mutable data structures, but unlike languages such as C, C++, or Java, they are not the default. Thus, programmers are encouraged to code purely functionally by default and to only resort to mutable data structures when absolutely necessary. In addition, unlike imperative languages, SML provides no support for mutable *variables.* In other words, the value of a variable cannot change in SML. Rather, all mutations must occur through data structures.

There are only two built-in mutable data structures in SML: refs and arrays. A value of type "int ref" is a pointer to a location in memory, where the location in memory contains an integer. It's analogous to "int*" in C/C++ or "Integer" in Java (but not "int" in Java). Like lists, refs are polymorphic, so in fact, we can have a ref (i.e., pointer) to a value of any type.

A partial signature for refs is below:

```
signature REF =
  sig
    type 'a ref

    (* create & initialize a new cell and
     * return the reference to it *)
    val ref : 'a -> 'a ref

    (* return the contents of the cell pointed
     * to by the ref *)
    val op ! : 'a ref -> 'a

    (* update the contents of the cell pointed
     * to by the ref *)
    val op := : 'a ref * 'a -> unit
  end
```

The following code shows an example where we use a ref:

```
  let val x : int ref = ref 3
      val y : int = !x
  in
      x := (!x) + 1;
      y + (!x)
  end
```

The code above evaluates to 7. Let's see why: The first line "val x:int ref = ref 3" creates a new ref cell, initializes the contents to 3, and then returns a reference (i.e., pointer) to the cell and binds it to x. The second line "val y:int = !x" reads the contents of the cell referenced by x, returns 3, and then binds it to y. The third line "x := (!x) + 1;" evaluates "!x" to get 3, adds one to it to get 4, and then sets the contents of the cell referenced by x to this value. The fourth line "y + (!x)" returns the sum of the values y (i.e., 3) and the contents of the cell referenced by x (4). Thus, the whole expression evaluates to 7.

Here's an example of a mutable stack using refs:

```
signature MUTABLE_STACK =
  sig
     type 'a mstack
     val new : unit -> 'a mstack
     val push : 'a mstack * 'a -> unit
     val pop : 'a mstack -> 'a option
  end

structure Mutable_Stack :> MUTABLE_STACK =
  struct
     (* a mutable stack is a ref pointing whose
      * contents contains a list of values *)
     type 'a mstack = ('a list) ref

     (* create a new ref cell and initialize it
      * with the empty list *)
     fun new():'a mstack = ref([])

     (* to push x on s, we cons x onto the current
      * contents of s, and then update the cell
      * referenced by s with the result *)
     fun push(s:'a mstack, x:'a):unit =
         s := x::(!s)

     (* top pop s, we dereference it to get the
      * contents, check for null (returning NONE)
      * and otherwise return the head of the list
      * after setting the contents of s to the tail. *)
     fun pop(s:'a stack):'a option =
         case (!s) of
           [] => NONE
         | hd::tl => (s := tl; SOME(hd))
   end
```

A good exercise for you is to implement a mutable version of queues, priority queues, dictionaries, or any other data structure that we've seen in class thus far using refs.

The other kind of mutable data structure that SML provides is called an array. Arrays generalize refs in that they are pointers to sequences of memory locations, where each location contains a different value. In some sense, we can think of a ref cell as an array of size 1. Here's a partial signature for the builtin Array structure for SML. Note that you have to "open Array" explicitly to use the operations or else write "Array.<foo>" when you want to call the <foo> operation.

```
signature ARRAY =
  sig
    type 'a array

    (* array(i,x) creates a new array of i elements all initialized
     * with the x. *)
    val array : int * 'a -> 'a array

    (* create an array from a list of values *)
    val fromList : 'a list -> 'a array

    exception Subscript

    (* get the ith element in an array -- raises
     * Subscript if the index is out of bounds *)
    val sub : 'a array * int -> 'a

    (* update the ith element in an array -- raises
     * Subscript if the index is out of bounds *)
    val update : 'a array * int * 'a -> unit

    (* return the length of the array *)
    val length : 'a array -> int

    ...
  end
```

See the SML documentation for more information on the operations available on arrays.