# CS 312 Recitation 15
# Refs and Arrays

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) other than printing. 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.

## Refs

There are only two built-in mutable data structures in SML: refs and arrays. SML supports imperative programming through the primitive parameterized `ref` type. 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

    (* ref(x) creates a new ref containing x *)
    val ref : 'a -> 'a ref

    (* !x is the contents of the ref cell x *)
    val op ! : 'a ref -> 'a

    (* Effects: x := y updates the contents of x
     * so it contains y. *)
    val op := : 'a ref * 'a -> unit
  end
```

A ref is like a box that can store a single value. By using the `:=` operator, the value in the box can be changed as a side effect. It is important to distinguish between the value that is stored in the box, and the box itself. A ref is the simplest **mutable** data structure. A mutable data structure is one that be changed imperatively, or **mutated**.

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 build using refs:

```
signature MUTABLE_STACK =
  sig
      (* An 'a mstack is a mutable stack of 'a elements *)
    type 'a mstack
    (* new() is a new empty stack *)
    val new : unit -> 'a mstack
    (* Effects: push(m,x) pushes x onto m *)
    val push : 'a mstack * 'a -> unit
    (* pop(m) is the head of m.
     * Effects: pops the head off the stack. *)
    val pop : 'a mstack -> 'a option
  end

structure Mutable_Stack :> MUTABLE_STACK =
  struct
    (* A mutable stack is a reference
     * to the list of values, with the top
     * of the stack at the head. *)
    type 'a mstack = ('a list) ref
    fun new():'a mstack = ref([])
    fun push(s:'a mstack, x:'a):unit =
        s := x::(!s)
    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 mutable versions of queues, priority queues, dictionaries, or any other data structure that we've seen in class thus far using refs.

## Arrays

Another important kind of mutable data structure that SML provides is the array. Arrays generalize refs in that they are a sequence of memory locations, where each location contains a different value. We can think of a ref cell as an array of size 1. The type `t array` is in fact very similar to the Java array type `t[]`. 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 use the operation ***`foo`***.

```
signature ARRAY =
  sig
    (* Overview: an 'a array is a mutable fixed-length sequence of
     * elements of type 'a. *)
    type 'a array

    (* array(n,x) is a new array of length n whose elements are
     * all equal to x. *)
    val array : int * 'a -> 'a array

    (* fromList(lst) is a new array containing the values in lst *)
    val fromList : 'a list -> 'a array

    (* indicates an out-of-bounds array index *)
    exception Subscript

    (* sub(a,i) is the ith element in a. If i is
     * out of bounds, raise Subscript *)
    val sub : 'a array * int -> 'a

    (* update(a,i,x)
     * Effects: Set the ith element of a to x
     * Raise Subscript if i is not a legal index into a *)
    val update : 'a array * int * 'a -> unit

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

    ...
  end
```

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

Notice that we have started using a new kind of clause in the specification, the **effects** clause. This clause specifies side effects that the operation has beyond the value it returns. When a routine has a side effect, it is useful to have the word "Effects:" explicitly in the specification to warn the user of the side effect.

## Substitution model and refs

The substitution model that we've seen so far explains how computation works as long as no imperative features of ML are used. This model describes computation as a sequence of rewrite steps in which a program subexpression is replaced by another until no further rewrites are possible. However, imperative features introduce the possibility of **state** : an executing ML program is accompanied by a current memory state that also changes as computation proceeds.

We don't want to get into the details of how memory heaps work yet, so we will use a simple abstract model of state. A memory $M$ is a collection of memory cells each with its own unique name. We will call these names **locations**; a location is an abstract version of a memory address at the hardware level. Given a location, we can look up in the memory what value is stored at that location. As the program executes, the contents of some memory locations may change.

One way to visualize the execution is the memory consists of a large (actually, infinite) number of boxes, each of which can contain a single value. At any given point during execution, some boxes are in use and others are empty. Each box has a unique name (its location) and this location can be used to find the single box with that name. Given a memory, we can always find a box that is unused.

## Ref operations

There are three principal operations on references: creation using the `ref` operator, deferencing using `!`, and update using `:=`. Each of these operations has an associated reduction that is used when evaluating it. In order to explain what these operations do, a new kind of expression is needed, representing a location. We will write the syntactic metavariable $loc$ to represent a location. For the purposes of explaining how to evaluate SML, we assume that there is an infinitely large set of locations (called $\mathbf{Loc}$) available for use when evaluating programs, even though the actual memory is infinite. We don't care what the elements of $\mathbf{Loc}$ actually are. We can think of them as memory addresses, as integers, or even as strings. All that matters is that we can tell two different elements of $\mathbf{Loc}$ apart.

### ref

The ref operation creates a new location. It is reduced once its argument is a value, creating a new location.

`ref` $v$ `-->` $loc$

The new location $loc$ is one that is unused in the current memory. This evaluation step also has a side effect: the memory cell named $loc$ is made to contain the value $v$.

This rule introduces a $loc$ expression into the running program. This is a bit different from all the evaluation rules that we have seen till this point, because a $loc$ expression cannot occur in the original SML program. This isn't a problem; we have to remember that our models of evaluation are useful fictions. As long as the model gives the right answer for what happens when the program runs, we are satisfied. In SML, if a program evaluates to a location, it is printed as a ref expression. However, note that two locations are not equal even if they have the same contents, because they are different locations:

```
- ref (2+2);
it = ref 4 : int ref
- it = ref 4;
val it = false : bool
```

### !

The dereference (`!`) operation finds the value stored at a given location:

`!` *loc* `-->` *v*

Of course, the value v that replaces the subexpression `!`*loc* is the value found in the memory cell named *loc*.

### :=

The update (`:=`) operation updates the value stored at a given location:

`!` *loc* `:=` *v* `-->` `()`

It evaluates to the unit value, but has the side effect of updating the memory location named *loc* to contain *v*.

## Example

Consider the following SML example:

```
let val x = ref 0
    val y = x
in
    x := 1;
    !y
end
```

What does this evaluate to? We can use the model about to figure it out:

```
let val x = ref 0
    val y = x
in x := 1; !y end
Memory: (empty)
-->
let val x = loc1      (loc1 is some new location)
    val y = x
in x := 1; !y end
Memory: (loc1 = 0)
-->   (substitute loc1 for x)
let val y = loc1
in loc1 := 1; !y end
Memory: (loc1 = 0)
-->   (substitute loc1 for y)
loc1 := 1; !loc1
Memory: (loc1 = 0)
--> !loc1
Memory: (loc1 = 1)
--> 1
Memory: (loc1 = 1)
```

So we see that the update to **x** is visible when we dereference **y**. This happens because **x** and **y** are **aliases** : two different names for the same location (***loc1***).