### Stateful Programming

Despite the virtues of pure functional programming, there are times when you'd like to be able to update memory directly.

Some examples:

- Generating unique identifiers.
- Manipulating a potentially cyclic graph structure.
- Building a histogram of values or doing a bin sort.
- Recording global state.

To support such applications, ML includes **references** and **arrays**.

### References

Think of a **reference** as a **container** for a value.

References are created using a special (built-in) data constructor `ref`, applied to the **initial contents** of the reference.

```
val a = ref 3
val a = ref 3 : int ref
```

The current value of a reference can be **fetched** using the `!` operator.

```
- val b = !a;
val b = 3 : int
```

The current value of a reference can be **updated** using the `:=` operator.

```
- a := !a + 4;
val it = () : unit
- !a;
val it = 7 : int
-
```

Note that the **type** of a reference is formed by applying the (built-in) parameterized type constructor $\alpha$ **ref** to the type are:

```
ref : 'a -> 'a ref
! : 'a ref -> 'a
op := : 'a ref * 'a -> unit
```

### Examples

We can use `ref`s for anything that an ordinary (imperative language) variable could be used for:

```
- val x = ref 0;
val x = ref 0 : int ref

- fun f () =
      if !x < 10 then
        (print (!x); print " ";
         x := !x + 1;
         f())
      else print "\n";
val f = fn : unit -> unit

- f ();
0 1 2 3 4 5 6 7 8 9
val it = () : unit
```

No real advantage over functional code in this case.

### Examples (cont.)

A better application: generating unique ids:

```
- local
  val counter = ref 0
in
  fun gensym () =
       (counter := !counter + 1;
        "sym" ^ (makestring (!counter)))
end;
val gensym = fn : unit -> string

- gensym();
val it = "sym1" : string
- gensym();
val it = "sym2" : string
```

### First-class references

References are fully first-class values in their own right. This means they can be embedded in data structures, passed as arguments or returned as function results.

For example, references can be used to get the effect of **call-by-reference** parameters.

```
- fun swap (x, y) =
   let val t = !x
   in x := !y;
      y := t
   end;
val swap = fn : 'a ref * 'a ref -> unit

- val a = ref 10;
val a = ref 10 : int ref
- val b = ref 20;
val b = ref 20 : int ref
- swap(a,b);
val it = () : unit
- !a;
val it = 20 : int
- !b;
val it = 10 : int
```

### Updateable dictionaries

```
type dict = (string * (int ref)) list ref
exception NotFound

fun empty ():dict = ref nil

fun lookup (x:string) (d:dict) : int =
   let fun f ((k,v)::r) =
              if k = x then !v
              else f r
        | f nil = raise NotFound
   in f (!d)
   end

fun update (x:string,y:int) (d:dict) : unit =
  let fun g ((k,v)::r) =
              if k = x then v := y
              else g r
       | g nil = d := (x,ref y)::(!d)
  in g (!d)
  end

- val a = empty();
val a = ref [] : dict
- update ("a",1) a;
val it = () : unit
- update ("b",2) a;
val it = () : unit
- update("b",3) a;
val it = () : unit
- lookup "b" a;
val it = 3 : int
```

### Incorrect use of references

Type system prevents us from making the mistake of trying to pass ordinary **values**:

```
- val c = 10;
val c = 10 : int
- val d = 20;
val d = 20 : int
- swap(c,d);
std_in:74.1-74.9 Error: operator and operand
                   don't agree (tycon mismatch)
  operator domain: 'Z ref * 'Z ref
  operand:         int * int
  in expression:
    swap (c,d)
```

### Arrays

There is a similar set of constructor and access functions for multi-element arrays, providing constant-time access and update to elements. To use arrays, must **first** say `open Array;`

Arrays are created **dynamically**; their size is fixed only when they are created, and is not part of their type. Array bounds are checked at runtime. Elements are indexed from 0.

```
- val a = array(100,0);
val a = [|0,0,0,0,0,0,0,0,0,0,0,0,...|] :
                                      int array

- update(a,1,9999);
val it = () : unit

- sub(a,0);
val it = 0 : int

- sub(a,1);
val it = 9999 : int

- sub(a,100);
uncaught exception Subscript
```

There is a similar datatype `Vector` which allows constant-time access but no update.

### Histograms

```
- fun bin l =
  let val a = array(10,0)
      fun inc x =
           let val n = x div 10
           in update(a,n,sub(a,n)+1)
           end
  in app inc l;
     a
  end
val bin = fn : int list -> int array

- fun display (a:int array) =
  let fun f n =
       if n < length a then
         (print n; print ":";
          print (sub(a,n)); print "\t";
          f (n+1))
       else print "\n"
  in f 0
  end

- display(bin ([1,4,66,3,45,77,45,66,88,9,20]));
0:4       1:0       2:1       3:0       4:2
5:0       6:2       7:1       8:1       9:0
val it = () : unit
```

### Doing without built-in state

What would it cost us to **insist** on being purely functional?

We can **simulate** the behavior of an $n$-element state at the cost of a factor of just $log_2(n)$.

To see this, view the reference mechanism as a way of associating unique **keys** (the reference itself) with **values** (the reference's contents).

• The `ref` operation invents a new key and pairs it with an initial value. (We cannot examine the key, but we can compare it with other keys!)

• The `!` operator fetches the value associated with a key.

• The `:=` operator changes the value associated with a key.

Viewed as an abstract datatype, this is just a **dictionary**. It is well-known how to implement dictionaries of size $n$ so that all operations take $O(lgn)$ time.

Arrays are similar, except that array index becomes part of key.