Mutable Data

Michael P. Fourman

February 2, 2010

References

Continuing our descent into anarchy, we introduce ML *references,* these are mutable, or updateable variables. If we allowed any value to be changed, at any point in the program, then the language would be like C, but with ML-style syntax. Furthermore, as we will see shortly, we would have to curtail the use of polymorphism. We really would have a mess on our hands.

The trick is to introduce a new kind of value, distinct from any of the values we have seen so far. Whereas the variables we have introduced so far refer to abstract mathematical values, a reference variable refers to a storage location in memory. (In this they are like arrays—an array may be thought of as referring to a structured block of storage which we can inspect using `sub`, and update, using `update`.)

Evaluating the expression `ref` e initialises a new updateable cell in memory with the result of evaluating e, and returns a value that we can use to refer to this memory location. We can bind this value to a name just like any other value:

```
> val r = ref (1 + (2 * 3));
val r = ref 7 : int ref
```

The type of the result is an `int ref`. An updateable cell is called a *reference* in SML; an `int ref` is a reference that can hold an integer. We can change the value in a reference cell using an assignment expression such as `r := 9+2`. The type of `:=` is `'a ref * 'a -> unit` and so the result of such an expression is just `()`. To retrieve the value within a reference we have to *dereference* it using the function `!`, as in the expression `!r + 8`. Note that a reference to an integer is *not* the same as an integer; evaluating `r + 8` would give you a type-checking error. You can have references of any type. You can even have polymorphic references, but with some restrictions. For example, suppose we typed `val rl = ref []` at the top-level. If this were allowed you could get into all sorts of problems. We could alter `rl` by evaluating `rl := [3]`. The type of `rl` will change from `'a list ref` to `int list ref`. A function that refers to `rl` may type-check before the change but may fail to type-check after the assignment. Here is an example of code that is **not legal in ML:**

```
val lp = ref []
fun test () = case !lp of [] => true
                  | (h :: t) => h = 0 ;
lp := [0.0, 3.2, 4.0];
test();
```

The type ML system rejects this code. If the code could be executed, the evaluation of `test()` would result in a comparison of a real and an integer. The same idea could be used to compare any two values. This is a type error, and would clearly be undesirable. The type of a reference cannot be allowed to change after its creation. Essentially this means that we can't create polymorphic references at the top-level.

Using references

References may be used in many ways. The most obvious is to mimic an imperative programming style. Here is an imperative implementation of the factorial function in ML.

```
fun fact n =
    let val count = ref n
        and prod  = ref 1
     in
        while !count > 0 do(
           prod  := !prod * !count;
           count := !count - 1
        );
        !prod
    end
```

This is similar to the corresponding C implementation

```
int fact( int n )
{
    int prod = 1;

    while(n > 0){
        prod = prod * n;
        n    = n - 1;
        }
        return prod;
    }
```

In ML, the one-line recursive definition is not only clearer, but also marginally faster.

Hidden state

Another use for references is to provide a mechanism for representing an updateable state when we want one. For example, a random number generator may be modelled as a “function” that returns a (pseudo-)random number. Unlike a true mathematical function, it should not always return the same value.

```
    functor MKRANDOM() =
    struct
      (* Given a seed, mkRandom returns a psuedo-random number generator
         which takes an integer argument of one more than the maximum
         return value required. ( Linear Congruential, after Sedgewick,
         "Algorithms", Addison-Wesley, 1983, Chapter 3 pp 37-38.) *)

      fun mkRandom seed =
            let val r = ref seed
        val a = 31415821
        val m = 100000011

        fun f n =
          let val rand = (r := ((a * !r + 1) mod m);
          (!r * n) div m)
          in if n < 0 then rand + 1 else rand
          end
             in f
            end;
    end;
```

Linked datastructures

References are used to build datastructures with loops. Consider the code

```
val ff = ref( fn 0 => 0 );

val ffact = fn 0 => 1
             | n => n * !ff (n-1);

ff := ffact;
```

Here, the recursive factorial function is implemented without explicit recursion. We use a reference as a place-holder for a function to be used in the recursive call, and then update this to make the code self-referential.

References may also be used to implement lists, trees and graphs. Sometimes operations on a mutable version of a data-structure may be implemented somewhat more efficiently than the corresponding functional structure. A mutable list can be implemented “by hand” as a linked list constructed with explicit references.

```
datatype 'a Node = Nil
                 | Cons of 'a * ('a Node ref)
type     'a List = 'a Node ref

fun insert (x:int) (xs : int List) =
    case !xs of
      Nil       => xs := Cons (x, ref Nil)
    | Cons(h,t) =>
        if x < h then (xs := Cons (x, ref (!xs)))
        else insert x t;

fun append (a: 'a List) (b: 'a List) =
    case !a of
      Nil => a := !b
    | Cons(h, t) => append t b;
```

Here is a series of interactions manipulating mutable lists.

```
> val a = ref Nil : int List;
val a = ref Nil : int List
> (insert 1 a; insert 3 a;insert 2 a);
val it = () : unit
> a;
val it = ref (Cons (1, ref (Cons (2, ref (Cons (3, ref Nil)))))) : int L
> val b = ref Nil : int List;
val b = ref Nil : int List
> (insert 10 b;insert 15 b;insert 17 b);
```

```
val it = () : unit
> append a b;
val it = () : unit
> a;
val it = ref
   (Cons(1,ref(Cons(2,ref(Cons(3,ref(
         Cons(10,ref(Cons(15,ref(
              Cons(17, ref Nil)))))))))))) : intList
```

Such implementations must be used with care. For example, `append a a;` produces a circular list, with no end. Many more arcane datastructures with more links have been developed. For example, doubly linked lists may be used to implement queues.

Hidden changes of representation

Some of the best uses of references are where they are hidden inside datatypes that look completely functional to the external user. For example, consider our 'two list' representation of a queue again. When we studied this example we noted that the second queue might be reversed many times. For example, if we had the variable `q` bound to the queue `Queue([], l)` then evaluating `front q` would result in the list `l` being reversed. However, this would occur *every time* we evaluated this expression, not just the first time. It would be better if we could destructively change the values of the two lists in this case to avoid such wasted effort. Here is an alternative version of the queue functor, that is based on references:

```
functor RQUEUE( type Item ):QueueSig =
struct
    exception Deq
    type Item = Item
    abstype Queue =
       Q of (Item list * Item list) ref
    with
        val empty         = Q (ref ([], []))
        fun isEmpty(Q (ref([], []))) = true
          | isEmpty _                = false

        fun enq(Q(ref (inp, out)), e) =
                Q(ref(e :: inp, out))
        fun deq(Q(ref(inp, h :: t))) =
               (Q(ref(inp, t)), h)
          | deq(Q(ref([], []))) = raise Deq
          | deq(Q(r as ref(inp, []))) =
               (r := ([], rev inp); deq(Q r))
    end
end;
```

Note how we can use `ref` as a constructor to avoid having to use `!` to peek inside a reference. Queue structures produced via this functor can be used in any context where a purely functional queue was required. The only difference would be in the execution speed. This example is slightly artificial, since queues are usually used in a "single-threaded" pattern of calls; we don't usually call `deq` more than once on the same queue.

A better example is given by using an array to automatically memoize values of a function. We combine two ideas: the tabulation of a function to avoid re-computation, and the use of update to implement recursion.

```
fun f memo 0 = 0
  | f memo 1 = 1
  | f memo n = memo (n-1) + memo (n-2);

infix 0 sub;
fun memoize n f =
let datatype R = K of int | U
    val a = Array.array (n, U)
    fun g x =
        (case a sub x of K n => n
                           | U   =>
         let val n = f g x in
             update(a,x,K n);
             n
         end
        )handle Subscript => f g x
in
    f g
end;
```

Memoizing the first 30 values of `fib` makes the computation of `fib 50` feasible.

```
>   val fib = memoize 30 f;
val fib = fn : int -> int
> fib 50;
val it = 12586269025 : int
```

For this example, the pattern of recursion is easy to understand, so we can write an efficient tail recursive function that keeps track of the previous computations it needs. For other functions with more complex patterns of recursion, memoization can be a useful tool.