Arrays

Michael P. Fourman

February 2, 2010

Introduction

We use the next two lectures to introduce some non-functional aspects of ML programming. We begin this note with more material on *exceptions,* and then go on to introduce *arrays.*

Trapping exceptions

We represent a partial function in ML by raising an exception when the function is called with an argument outside its domain. We give two examples: evaluating the expression `1.0/0.0` raises the exception `Quot`; if a key, `k`, has no entry in the dictionary, `d`, then evaluating `lookup d k`, raises the exception `Lookup`.

These exceptional cases are not always errors; sometimes they are just cases that require special treatment. We want to write code that will handle such cases gracefully. One way is to test for exceptional cases explicitly, using pattern-matching or a conditional, before applying the function. Here is an example:

```
fun ratio x = if x = 0.0 then 1.0 else sin x / x
```

The effect is to return the value 1.0 in the cases where evaluating `sin x / x` would raise the exception `Quot`.

ML provides another way: we can apply the function, and *handle* the exception if it is raised. We code our example using this technique:

```
fun ratio x = sin x / x handle Quot => 1.0
```

The effect is similar but the mechanism is subtly different; in this case, we return the value 1.0 in the cases where evaluating `sin x / x` *does* raise the exception `Quot`.

Let e be the expression $e \equiv exp_1$ **handle** exn `=>` exp_2. To evaluate e, we first evaluate the expression exp_1. If this evaluation raises no exception, we

return the result as the value of e. Otherwise, if the evaluation of exp_1 does raise an exception, one of two things can happen: if the exception is matches *exn* then the expression exp_2 is evaluated and the result (if any) is returned as the value of e; if the exception raised when evaluating exp_1 is different from *exn* then the exception is propagated—it acts as if there were no handler. Thus a raised exception bubbles up the call stack until it either reaches the top (and is reported at the top level) or else bumps into a handler willing to handle it.

In our example the potential source of an exception is (syntactically) close to the handler. However, a handler will be applied to any exception raised by the evaluation of the expression it follows, even if the exception was raised by a call to some auxiliary function elsewhere in the code.

Exceptions can have values associated with them, as illustrated by the `Error` exception, introduced by the declaration

```
exception Error of string;
```

The names introduced by exception declarations are constructors (they construct values of type `exn`); they can be used in patterns, just like other constructors. A handler can use a pattern in the form

exp_1 `handle` *pat* `=>` exp_2.

to retrieve the value associated with the exception and use it in exp_2.

The general form of a handler is

exp_1 `handle` *match.*

A *match,* may contain multiple clauses

pat_i `=>` exp_i

separated by `|`. The syntax is just like that used in anonymous functions (`fn`), and `case` expressions.

Our signature for a queue provides a test `isEmpty` so we can test if a queue is empty before attempting to apply the function `deq`, which fails on an empty queue. Our signature for a dictionary provides no test to check is an entry exists for a given key; unless we can guarantee that we will only lookup valid keys, we must provide a handler for the exception `Lookup`.

To look up a string, `s`, in a dictionary, `d`, we evaluate the expression, `lookup d s`. This may fail by raising the exception `Lookup`. Suppose that the dictionary associates an expression with a string, and that the appropriate expression to use for a string not entered in the dictionary is an identifier formed from `s` by applying a constructor `Id`. We can handle the exception by returning this value:

```
fun find d s = lookup d s handle Lookup => Id s;
```

1 Arrays

In functional programming, we often use a list where a C programmer might use an array. Lists have advantages: they are dynamic datastructures, we do not need to decide in advance how large to make a list; they are immutable datastructures, so a functional style can be used. However, arrays are well-matched to conventional machine architectures, and are fast.

The system predefines a structure `Array` matching the signature

```
sig eqtype 'a array     (* this means it is an equality type *)
    exception Subscript (* for subscripting errors  *)
          and Size      (* for arrays with -ve size *)
    val array       : int * 'a -> 'a array
    val arrayoflist : 'a list -> 'a array
    val sub         : 'a array * int -> 'a (* constant time indexing *)
    val update      : 'a array * int * 'a -> unit    (* and updating *)
    val length      : 'a array -> int
    val tabulate    : int * (int -> 'a) -> 'a array
end
```

An array, a, is a data-structure that contains a collection of entries a_i. Given an integer *index*, i, the expression a `sub` i returns the entry a_i in constant time. (We use `sub` as an infix, with precedence 0.) Each array has a fixed length, $n \geq 0$. Arrays are indexed from 0, so an array of length n has an entry for each index $i \in \{0, \ldots, (n-1)\}$.

Here is a function to create a list of the entries in an array

```
fun listOfArray a =
let fun listFrom n = if n = length a then []
                     else (a sub n):: listFrom (n+1)
in
    listFrom 0
end;
```

Arrays may be created in three ways: the function `array` creates an array of entries that all have the same value; the function `tabulate` creates an array with the entry a_i given as a function of the index i; and the function `arrayOfList` creates an array whose entries are the members of some list.

Arrays are *mutable* datastructures; the entries in an array may be changed, or updated. The function `update` changes the value of a given entry. Its purpose is to make this change, rather than to return a value; because it is a function, it must return some value, so it returns the value `()` which is the only value of a type called `unit`. In ML, functions that return a value of type unit are used like procedures in other languages.

Sometimes we want to call a sequence of procedures. In ML we can form an expression from a sequence of expressions, enclosed in parentheses and separated by semi-colons; to evaluate this new expression we evaluate each expression in the sequence, in turn, and return the value of the last one. Here is an example, giving the ML code to swap to entries in an array.

```
fun swap (a, i, j) =
    let val t = a sub i
    in (
        update(a, i, a sub j);
        update(a, j, t)
       )
    end
```

Arrays as Functions

We may use an array to represent a function. Given a function `f: int -> A`, the call `tabulate (n, f)` creates an array giving the first n values of `f`. We can use the array to access these values in constant time. If we know that the first n values of `f` (for arguments $0 \ldots (n-1)$) will be required frequently, we can compute them, once and for all, and store the values in an array. The function `f` can be implemented by looking for values in this array, and only *computing* results for arguments outside the range of the array.

```
val f = let val a = tabulate(n,f)
         in
            fn x => (a sub x handle Subscript => f x)
        end
```

Here, we trade space for speed. We compute the values of f up front, them we can use them as many times as we like (almost) for free. A slightly more sophisticated version of this technique, called *memoization,* because we make a note, or memo, of the values we compute up front, will be introduced in the next lecture.

Arrays as sets

An array can be used, just like a list, to represent a set. The elements of the set are the entries in the array. The representation is made more flexible if we consider the sets represented by array segments. A triple, (l, a, u), of type `int * A array * int` can be used to represent the set, $\{a_i | l \leq i < u\}$, of all entries with indices between some lower bound, l, and upper bound, u.

Arrays come into their own when we want to represent a number of disjoint subsets of a given set, and move elements between them. Consider the problem of producing a random permutation of the first n integers $(0 \ldots n)$. We can place the integers in a set, and make n random selections from the set. We remove each element from the set as it is selected, and place it at the beginning of a sequence representing the permutation. This algorithm is efficiently implemented using a single array, a, to represent both the set of remaining integers, and the permutation as it is built up. We use a function `rand` to make our random selections; `rand n` returns a number randomly chosen from $\{0, \ldots, (n-1)\}$.

We use a number, s, to separate two parts of the array. The set is $\{a_i | i < s\}$; the permutation of the elements removed so far is the sequence $\langle a_i \rangle_{s \leq i < n}$. At each step of the algorithm, we select a member of the set at random, swap it with the entry a_{s-1} (which changes neither the set, nor the permutation), and then reduce s by 1 to move it from the set to the permutation.

```
functor RPerm(ArraySig) =
struct
fun rperm n =
let val a = tabulate (n, fn x => x)
    fun swap(i,j) = let val t = a sub i
                    in( update(a,i,a sub j);
                        update(a,j,t))
                    end
    fun permute 0 = ()
      | permute s = (swap(rand s, s-1);
                     permute (s-1))
in
    (permute n; a)
end
```

This example demonstrates arrays to best advantage. Using an array provides a compact and flexible representation with constant time access to a randomly chosen element. Hoare's quicksort algorithm, which we will introduce later in the course, uses an array in a similar way to represent a number of sets.

In the next section we discuss another way to represent sets using arrays.

Hashing

Arrays have two serious disadvantages. First, and most serious, they are mutable; the value of an expression involving an array may change. Controlling such changes is difficult. In the next lecture we will look in more detail at mutable data structures, and see how we may combine some of the benefits of arrays with the safety and tractability of immutable data.

The second disadvantage is that an array represents a function defined on an initial segment of the natural numbers. We can use arrays to implement sets, graphs, dictionaries, and other datastructures. Using arrays naively may require arrays of an impractical size, furthermore, the amount of data actually stored may be small in comparison to the size of the array. *Hashing* is a technique used to overcome these problems.

We introduce hashing using a dictionary as an example, and then see how the same ideas may be applied to the implementation of sets. We can use an array to provide an efficient implementation of a dictionary.

```
functor NaiveDict(type Item) =
   struct local open Array in
       type Key       = int
       datatype Entry = None | Some of Item
       type Dict   = Entry array
       exception Lookup

       fun empty n = array(n, None)

       fun lookup d n =
       (case d sub n of None => raise Lookup
                      | Some item => item)
        handle Subscript => raise Lookup

       fun enter((k,e),d) =
                           update(d, k, Some e)
       fun remove(k,d)    = update(d, k, None)
   end
end;
```

Notice that this is a mutable dictionary; `enter` and `remove` are procedures, not functions that return a modified dictionary. This naive implementation restricts us to integer keys, and dictionaries of known size. Furthermore, if the keys used are sparsely distributed much space is wasted.

We will use hashing to provide another implementation of arrays. The

basic idea is to define a simply-computed *hash function*, h, that maps each key to an index into an array called a *hash table* which contains the stored items. There are many variations on this idea; we will examine one. A data-structure containing entries for all the keys that hash to a particular index is stored, under that index, in the array.

```
functor HASHDICT( structure D:DictSig
                  val hash: D.Key -> int
                  val size:int) =
  struct local open Array in
    type Dict = D.Dict array
    val Lookup = D.Lookup
    fun empty () = array(size, D.empty)

    fun lookup d k =
                   D.lookup (d sub (hash k)) k

    fun enter((k,e),d) =
        let val h = hash k in
        update(d, h, D.enter((k, e), d sub h))
        end

    fun remove(k,d) =
        let val h = hash k in
        update(d, h, D.remove(k, d sub h))
        end
  end
end;
```

This functor produces a mutable data-structure supporting the dictionary operations. The hash function `hash` should return an integer h, with $0 \leq h <$ `size`, for each key. To access an entry, we use the hash function and array operations to access the appropriate dictionary (in constant time). The time taken to access the entry will depend on the size of this dictionary. A hash function is *efficient* if it distributes entries evenly over the array. Suppose our hash function is efficient, that the number of items stored is N, and that the size of the hash table is n, then the expected number of items stored in each dictionary will be N/n. Provided we can always construct efficient hash functions, we can trade space for time by varying the size of the hash table.

If we use an association list to provide our basic implementation of a dictionary, this technique is called *linear chaining* because the data for items hashing to a given index is in a list, or chain that is accessed via the hash

table. A similar approach could be used to produce a mutable data-structure supporting operations on sets.

Hash functions may be used to map a large range of keys to a smaller range of indices. For example, a simple hash function for integer keys is simply

```
fun hash n = n mod size
```

For uniformly distributed keys this will work well. A great deal of attention has been paid to the design of hash functions for applications using strings as keys. This is still a black art. Experience seems to show that it is a good idea to use a prime for the size of the hash table, and it is normally worthwhile making the hash function depend on all the characters in a string. Variations on the following function are common

```
val size = 211 (* a prime*)
local fun combine [] = 0
        | combine (h :: t) = (ord h + 7 * combine t) mod size
in
      fun hash s = combine (explode s)
end;
```

The only real test of a hash function is to apply it to a sample set of data, from the intended application.