Examples of Imperative Programming in F#

Torben Mogensen

October 14, 2016

These notes show some simple examples of imperative programming to supplement the notes and textbook.

1 Example 1: Using a global counter to count function calls

We start from the doubly-recursive version of Fibonacci's function:
et rec fib n = if n ¡ 2 then n else fib (n-1) + fib (n-2)
We can call this to find, for example `fib 10` $\rightsquigarrow$ `55`. But how many function calls does this version of `fib` use?

To figure this out, we use a mutable variable `counter` and modify `fib` to increment this for every function call and, finally, print the result out:

```
let fib n =
  let mutable counter = 0
  let rec fib' n =
    counter <- counter + 1
    if n < 2 then n else fib' (n-1) + fib' (n-2)
  let result = fib' n
  printfn "fib used %d function calls" counter
  result
```

When we call `fib 10`, it will print "`fib used 177 function calls`" before returning the value (which is still 55). Note that the call to `fib'` needs to come before the print statement, but it is the value of this call that must be returned. Hence, we bind the value in `result`, so we can return it after the print statement.

We can use such counters to get a rough estimate of the time used by a function. We can, for example, compare to the version of fib that uses accumulating parameters:

```
let fibA n =
  let mutable counter = 0
  let rec fib' n a b =
    counter <- counter + 1
    if n = 0 then a else fib' (n-1) b (a+b)
  let result = fib' n 0 1
  printfn "fibA used %d function calls" counter
  result
```

When calling `fibA 10`, it prints "`fibA used 11 function calls`" before returning 55. Since there is not signifantly more work done per function call, the accumulating version is much faster.

2 Example 2: Imperative Functions on Arrays

Array elements are mutable, so it can sometimes be useful to use imperative programming when working with arrays, especially if using library functions like `Array.init`, `Array.map`, `Array.filter`, and `Array.fold` are not applicable to the situation. For example, the Sieve of Erastothenes method for finding primes (as shown in previous notes) is not easily written using the library functions.

Imperative programming is also useful when working on arrays that are so large that creating a new array of similar size using, say, `Array.map`, can use more memory than available. With Imperative programming, we can re-use the original array and write the result to this instead of creating a new array. We can no longer use the old array, but we have used less memory during the operation. But programming *in-place* (without using extra space) can be more difficult than when we are allowed to use extra space.

Let us consider a function that takes an array of numbers and moves all the odd numbers to the beginning of the array and all the even numbers to the end, but keeping the relative positions of the odd and even numbers. For example, the array `[|1;2;3;8;7;5;4|]` should be replaced by `[|1;3;7;5;2;8;4|]`. The function should have the type

```
oddEven : int [] -> int []
```

Doing this functionally using library functions is easy:

```
let oddEven a =
  Array.append (Array.filter (fun n -> n%2 = 1) a)
               (Array.filter (fun n -> n%2 = 0) a)
```

If we are allowed to create a new array, doing it imperatively is not that difficult either (albeit slightly more verbose):

```
let oddEven' a =
  let ll = Array.length a
  let newA = Array.create ll 0  // make new array
  let mutable j = 0
  for i = 0 to ll - 1 do // copy odd elements to new array
    if a.[i] % 2 = 1 then
      newA.[j] <- a.[i]
      j <- j + 1
  for i = 0 to ll - 1 do // copy even elements to new array
    if a.[i] % 2 = 0 then
      newA.[j] <- a.[i]
      j <- j + 1
  newA // return new array
```

Note how `j` keeps track of where the copied elements go.

Now, let us consider doing this in-place, i.e., without allocating a new array, but leaving the result in the same array as the input. We want a function of the type

```
oddEvenI : int [] -> unit
```

We can not from the type see that the input array is modified, but we can see that the result type is `unit`, so the function does not return a new array.

Since we are not allowed to create a new array, we must work by re-arranging elements in the old array, which we can do by swapping elements. One possible strategy is:

1. Start from position $p = 0$.

2. Locate the first *even* element at some position $i \geq p$, if any.

3. If there are none, we are done, as the array starting from p is empty or all elements in the array are odd.

4. Locate the first *odd* element at some position $j > i$, if any.

5. If there is no such element, we are done, because there are no odd elements after the first even element.

6. We now have even elements in positions i to $j - 1$ and an odd element in position j. We can get the odd element at position j in place by putting it in position i, but then we must slide the even element in position i to position $i + 1$, and so on, until the even element in position $j - 1$ is put into position j.

7. Set p to $p + 1$ and repeat from step 2.

We can illustrate this on the array `a = [|1;2;3;8;7;5;4|]`:

i. We start by $p = 0$ and find $i \geq p$ such that `a.[i]` is even. This makes $i = 1$, since `a.[1] = 2`.

ii. We now find $j > i$ such that `a.[j]` is odd. This makes $j = 2$, as `a.[2] = 3`.

iii. We now put 3 into place 1 and 2 into place 2, making `a = [|1;3;2;8;7;5;4|]`.

iv. We set p to $i + 1 = 2$, set i to the position of the next even element, so $i = 2$, since `a.[2] = 2`.

v. We now find $j > i$ such that `a.[j]` is odd. This makes $j = 4$, as `a.[4] = 7`.

vi. We now put 7 into `a.[2]` and slide the elements in position 2 to $4 - 1$ one position up, making `a = [|1;3;7;2;8;5;4|]`.

vii. Repeating this once more brings 5 into place, so we get `a = [|1;3;7;5;2;8;4|]`.

viii. Since there are no odd elements after the 2, we are now done.

We can code this strategy in F# like this:

```
let oddEvenI a =
  let ll = Array.length a
  let mutable p = 0
  while p < ll do
    let mutable i = p
    while i < ll && a.[i] % 2 = 1 do // find even element
      i <- i + 1
    if i < ll then  // if not done, then
      let mutable j = i + 1
      while j < ll && a.[j] % 2 = 0 do  // find odd element after that
        j <- j + 1
      if j < ll then // if not done, then
         // odd number at j is put into place and even elements slide up
        let mutable elem = a.[j]
        while i <= j do
          let tmp = a.[i]
          a.[i] <- elem
          elem <- tmp
          i <- i + 1
    p <- p + 1  // and repeat with larger p
```

This is not only very verbose, it is also less efficient: Where the previous two versions use time linear in the size of the array, this version can in the worst case need quadratic time: Consider an array with even numbers in position 0 to $n - 1$ and odd numbers in position n to $2n - 1$. Each iteration will slide all n even numbers one position up, and n iterations are needed, so the total time used is proportional to n^2.

But it does use less memory.

Besides being examples of imperative programming, the lesson in this is that doing array operations completely in-place can seriously complicate programs and even make them run slower. But when operation on *very* large arrays, it may be neccesary.