Imperative programming in F#

Prakash Panangaden

26th January 2017

So far we have used the *functional paradigm* exclusively. In this framework the basic entities are *expressions* and *values*. Values are special expressions that are the end result of a computation. The process of changing a general expression to a value is called *evaluation*. No values were ever modified. For example, when we sorted a list, we actually created a new, sorted *copy* of the original list. The original list remains in its unsorted form. One may cavil at the concomitant inefficiency but the fact remains that it makes programming very easy.

We may associate a name with a value, for example by using **let**. Such a correspondence is called a *binding*. The expression where a binding is in force is called its *scope*. The correspondence between names and values is called an *environment*. We will study environments in detail next week.

It is undeniable, however, that the ability to modify data is vital: not just for efficiency but also *conceptually* one is often modelling something that is changing *in time* and one needs to make updates to reflect this. Certainly most simulation programs are like this. In these notes we will look at how imperative features are incorporated into F#.

Now we look at *mutable variables*. These are variables that can be updated in the way that you are used to in other languages. There is a fundamental new *semantic* entity:

COMMANDS.

A command or *instruction* is an order to "do something". The basic command is to change the value of a variable. You are, of course, familiar with this from your prior experience, but we will take a closer look at what it means.

First we introduce a new kind of data: a *location*. This is supposed to be an abstract picture of a piece of memory, hence memory location. Concretely it is the *address* of something in memory which can be modified. We will not use actual machine address, we will imagine that we have access to an unlimited supply of memory cells called **locations**. Ideally it should be thought of as a new type constructor and in SML, OCAML and Haskell it is introduced like that but in F# it is introduced in a slightly confusing way. That is a (minor) design mistake but we can live with it. Here is how you indicate that a name denotes a location rather than a value:

```
>let mutable x = 1
val mutable x : int = 1
```

It says `x` is mutable and of type integer. What this means is that `x` is the *name of a memory cell that stores integers*. Remember **`x` is not 1**!! `x` is the name of a memory cell that happens to store the value 1 *at that time*. We can change the value stored inside the cell **but x always means the same cell**. Here is a sample script.

```
> x;;
val it : int = 1
> x <- 2;;
val it : unit = ()
> x;;
val it : int = 2
```

You see how we have changed the value stored in the cell but the association of `x` and the cell has not changed. We used the word "environment" to refer to the correspondence between names and values. We continue to do so but now we include locations as values. We introduce a new mapping: the store is a map from locations to values. We can update the store using assignment statements but we cannot update a binding. We can create and destroy bindings by entering and exiting new scopes or making function calls, but we cannot rebind the *same* name to a *new* value. Stores, however, *are* updatable.

Here are some examples which I show without further explanation; they should be clear.

```
> let mutable lst = [1;2;3;4];;

val mutable lst : int list = [1; 2; 3; 4]

> lst <- [5;6;7];;
val it : unit = ()
> lst;;
val it : int list = [5; 6; 7]
> let mutable f = cos;;
val mutable f : (float -> float)
> f 1.57;;
val it : float = 0.0007963267107
> f <- sin;;
val it : unit = ()
> f 1.57;;
val it : float = 0.9999996829
> let mutable (u,v) = (2,"abc");;
val mutable v : string = "abc"
val mutable u : int = 2
> v <- 3;;

imp_examples.fs(16,6): error FS0001: This expression was expected to have type
    string
but here has type
    int
> v;;
val it : string = "abc"
```

```
> v <- "foo";;
val it : unit = ()
> v;;
val it : string = "foo"
>
```

The basic update command has the form: `exp1 <- exp2`. The evaluation rule for this is as follows:

1. First evaluate `exp1` and verify that the result is a location.
2. Then evaluate `exp2` and verify that the *value* obtained has the type appropriate to the location. Note, what gets stored are values, you **cannot** store unevaluated expressions.
3. **Replace** the contents of the location from step 1 with the value in step 2.

This is familiar to you but I want to bring two points to your attention: (i) an assignment *destroys* an old value, thus the programmer has control over (and *responsibility* for) the *lifetime* of data. She gets to decide whether a value is needed any more and makes a *choice* to reuse a storage cell. This is what we could not do with functional programming. (ii) The name of a variable means *two different things* depending on where it appears in an assignment statement. If we write `x <- x + 1`, the `x` on the left of the assignment symbol means "the location denoted by `x`" whereas the one on the right means "the value stored in the location."

Suppose that `x` is bound to location $l0$ and this location contains 11. Then the *execution of the command* `x <- x + 1` proceeds in the following stages, $C(\cdot)$ means "contents of":

Command	Environment	Store
`x <- x +1`	$x \mapsto l0$	$l0 : 11$
$l0$ `<-` $C(l0) + 1$	$x \mapsto l0$	$l0 : 11$
$l0$ `<-` $11 + 1$	$x \mapsto l0$	$l0 : 11$
$l0$ `<-` 12	$x \mapsto l0$	$l0 : 11$
Done	$x \mapsto l0$	$l0 : 12$

We do not need any special syntax for contents of in the language, it is automatic for names on the right hand side of assignment statements: this is called automatic *dereferencing.*

What about using functions with mutable values? Well here is what happens:

```
> let modify n = n <- n + 1;;
imp_examples.fs(5,16): error FS0027: This value is not mutable
```

Oh, should I have declared n to be mutable?

```
> let modify (mutable n: int) = n <- n + 1;;
imp_examples.fs(6,13): error FS0010: Unexpected keyword 'mutable' in
pattern.
>
```

Hmmm, can we use mutable values inside functions? In the following `x` is declared mutable and has the value 2 initially.

```
let mess_with () = x <- x + 1
val mess_with : unit -> unit
> x;;
val it : int = 2
> mess_with ();;
val it : unit = ()
> x;;
val it : int = 3
```

What about local variables?

```
let munge (n:int) =
  let mutable m = n
  (m <- m + 1);;
```

We can write this but it is absolutely useless; what happens to m is hidden to the outside world. When `munge` exits the local bindings are thrown away so any changes to m, indeed the very existence of m is invisible to the outside. Now if `x` is a declared mutable and global to a function definition we can see effects. What we need is some way to see the effects. This brings me to another topic; we will return to our question about mutable local variables presently.

A command is viewed as a special kind of expression that returns `unit`. How do we do several commands in a row? This is *sequential composition* and is done with a semicolon (just one). If you have a sequence of commands ending with an expression, the last value is returned. Here is an example

```
let result = (printfn "x is %i" x); (mess_with ()); x
> x is 3

val result : int = 4
```

The first line is what I typed, the rest was echoed by the interpreter. The first line contains two commands followed by an expression. The commands do not return a value but they have a *side-effect*. Here we are using global variables.

Now back to our question with local variables. Can we use print to show that mutable variables are being changed?

```
let mash (n:int) =
  let mutable m = n
  (printfn "m is %i" m);(m <- m + 1); (printfn "n is %i" n);
  (printfn "m is %i" m); m
> mash 5;;
m is 5
n is 5
m is 6
val it : int = 6
```

Can we write full-blown imperative programs with `while loops` and all those other things that you have been pining for? If *b* is a boolean expression and *e* is an expression of any type then `while` *b* `do` *e* is an expression of type `unit`. It works as you might expect but it always returns unit even if *e* has some other type.

```
let foo n =
  let mutable x = n in
    while (x < 10) do (printfn "x is %i" x);(x <- x + 1)
> foo 5;;
x is 5
x is 6
x is 7
x is 8
x is 9
val it : unit = ()
```

It is frustrating however that we cannot pass mutable values as arguments. Surely that is too limiting? Yes, and we can but not with simple values. We must use records. I expect you to learn about record syntax and fields on your own but here is a simple example from which you can learn the syntax.

```
type Person = { name : string ; birthday : int * int; title : string }

let prakash = { name = "Prakash"; birthday = (3,11);
                title = "Bane of while loops"}
> prakash.name;;
val it : string = "Prakash"
```

OK, now for *mutable* records. We can declare records with mutable fields and these can be passed to a function.

```
type intRec = { mutable count : int }

let r1 = { count = 0}
val r1 : intRec = {count = 0;}
> r1.count <- 1729;;
val it : unit = ()
> r1;;
val it : intRec = {count = 1729;}

> let increment (counter: intRec) =
    counter.count <- counter.count + 1
    counter.count;;
val increment : counter:intRec -> int
> increment r1;;
val it : int = 1730
> increment r1;;
val it : int = 1731
```

Why allow this with records and not with plain mutable variables? I will explain that later. It is an F#-specific design decision not followed, for example, in SML.

If you really want just a single variable as a mutable variable that can be passed in as an argument, you have to make a record with just one field. This is so common that F# provides a shorthand to do that. This shorthand makes it look very much like how mutable variables are defined in other functional languages like SML (but not like Haskell).

We define a **reference** type to be a record type with one mutable field called `contents`. Thus if we were to write `type counter = int ref` it is equivalent to writing
`type counter = { mutable contents : int}`.

Thus, we have the following equivalences:

```
type 'a ref = { mutable contents: 'a }
let ref v = { contents = v }
```

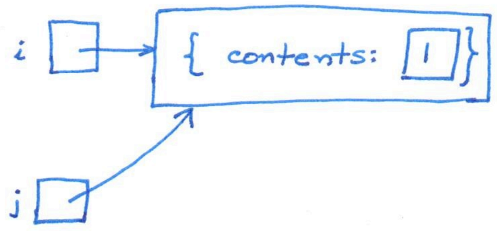
If we declare `let x = ref int`, we access the data by writing `x.contents` but we can write this as `!x`. We can update the contents by writing `x := x + 17` instead of `x.contents <- x.contents + 1`. These are just shortcuts but they make the type system look as if ref types are a primitive type constructor. It is useful to conceptualize them that way.

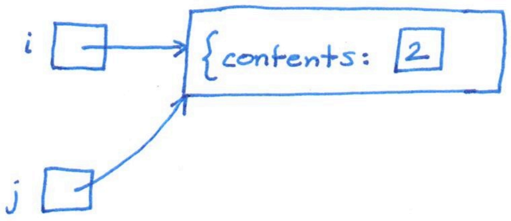
```
>type counter = int ref
type counter = int ref
>let i = ref 0;;
val i : int ref = {contents = 0;}
>let increment (c : counter) = c := !c + 1
val increment : c:counter -> unit
>increment i
val it : unit = ()
> i;;
val it : int ref = {contents = 1;}
```

What happens if we write `let j = i`? We get a phonomenon called *aliasing* where two names refer to the same data.

```
> let j = i;;
val j : int ref = {contents = 1;}
> increment i;;
val it : unit = ()
> j;;
val it : int ref = {contents = 2;}
```

The picture below shows what happened.





When you declare a record like `i`; the name i is associated with a *pointer or reference* to the actual record. So the box describing x does not contain the record; it contains the *address* in memory where the record actually resides. When the binding `let j = i` is done a *new* name j is created and it is bound to the result obtained by evaluating i. This value is the address of the record, so the same address is associated with j. Now both i and j are **aliases** for the same record. Any update to i will automatically affect j. *Be very careful* when writing imperative code to avoid unintended aliasing.