Imperative programming in OCaml

Prakash Panangaden

1st February 2020

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 have already studied the environment model. We quickly recall the basic terms here. 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*.

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 OCaml.

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**. Here is how you indicate that a name denotes a location rather than a value:

```
# let x = ref 1;;
val x : int ref = {contents = 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*. More precisely; it is the name of a record with a single *mutable* field called `contents` in which, *for the moment*, the value 1 is stored. Remember **`x` is not 1**!! `x` is the name of a record with a mutable field that happens to store the value 1 *at that time*. We can change the value stored inside the mutable field **but x always means the same record**. We can retrieve the value from a `ref` by using the ! symbol as shown below; this symbol is called the *dereferencing operator*. We update a value by using the symbol `:=`. Here is a sample script.

```
# let x = ref 1;;
val x : int ref = {contents = 1}
# x;;
- : int ref = {contents = 1}
# x := 2;;
- : unit = ()
# x;;
- : int ref = {contents = 2}
# !x;;
- : int = 2
# let y = ref 2;;
val y : int ref = {contents = 2}
```

You see how we have changed the value stored in the record but the association of `x` and the record 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.

Note how the keyword `ref` is used in two ways: as a **type constructor** so `int ref` is a *new type different from* `int`; and as a data constructor so that `ref 1` creates a new kind of data value.

In the presence of updatable values equality becomes subtle. Continuing the script from above we see:

```
# x = y;;
- : bool = true
# y := 3;;
- : unit = ()
# x = y;;
- : bool = false
```

There is another equality symbol `==` which stands for physical equality.

```
# let u = ref 7;;
val u : int ref = {contents = 7}
# let v = u;;
val v : int ref = {contents = 7}
# !u;;
- : int = 7
# !v;;
- : int = 7
```

```
# u = v;;
- : bool = true
# v := 8;;
- : unit = ()
# !v;;
- : int = 8
# !u;;
- : int = 8
# let w = ref 8;;
val w : int ref = {contents = 8}
# u = w;;
- : bool = true
# u == w;;
- : bool = false
```

The binding `let v = u` says bind `v` to the expression on the right hand side of the `let`, i.e. to the *value of u* which is the mutable record to which the name `u` is bound. Thus `u` and `v` now denote the same mutable data and an update to `v` also affects `u`; as we see above. We say that `u` and `v` are *aliases*. *Be very careful* when writing imperative code to avoid unintended aliasing. When we declare `w` we create a new record which happens to contain 8 so the basic equality test = will say `true` for `u = w` but the physical equality test == will say `false` since these are different records.

For brevity, I will henceforth say "cell" rather than "record with mutable field".

```
# u := u + 1;;
Characters 5-6:
  u := u + 1;;
       ^
Error: This expression has type int ref
       but an expression was expected of type int
# u := !u + 1;;
- : unit = ()
# !u;;
- : int = 9
# !v;;
- : int = 9
```

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) In a conventional programming language 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." In OCaml we use the notation `x := !x + 1` so that the conversion of a location to a value is made explicit; in this way `x` *always means the same thing.*

Suppose that `x` is bound to location $l0$ and this location contains 1729. 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`	$x \mapsto l0$	$l0 : 1729$
$l0$:= $C(l0) + 1$	$x \mapsto l0$	$l0 : 1729$
$l0$:= $1729 + 1$	$x \mapsto l0$	$l0 : 1729$
$l0$:= 1730	$x \mapsto l0$	$l0 : 1729$
Done	$x \mapsto l0$	$l0 : 1730$

Can we store other structures in updatable form? Yes we can!

```
# let l = ref [1;2;3];;
val l : int list ref = {contents = [1; 2; 3]}
# l := 0::!l;;
- : unit = ()
# !l;;
- : int list = [0; 1; 2; 3]
# let l2 = [0;1;2;3];;
val l2 : int list = [0; 1; 2; 3]
# l = l2;;
Characters 4-6:
  l = l2;;
      ^^Error: This expression has type int list
       but an expression was expected of type int list ref
# !l = l2;;
- : bool = true
# !l == l2;;
- : bool = false
```

We can create records with more than a single mutable field. To update a mutable field one uses the operator `<-` as shown below. One can define functions that have a side effect; such functions return `()` which is the unique value of the special type `unit`.

```
# type point = {mutable x: int; mutable y: int};;
type point = { mutable x : int; mutable y : int; }
# let p = {x=3; y = 4};;
```

```
val p : point = {x = 3; y = 4}
# p.x;;
- : int = 3
# p.x <- 5;;
- : unit = ()
# p;;
- : point = {x = 5; y = 4}
# let move (p:point) (a,b) = (p.x <- p.x + a);(p.y <- p.y + b);;
val move : point -> int * int -> unit = <fun>
# move p (2,5);;
- : unit = ()
# p;;
- : point = {x = 7; y = 9}
```

Why use `:=` for references and `<-` for mutable fields? References are really records with a single mutable field. We can use the following if we like:

```
# x;;
- : int ref = {contents = 5}
# x.contents <- 17;;
- : unit = ()
# x;;
- : int ref = {contents = 17}
```

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

```
# let modify n = n:= !n+1;;
val modify : int ref -> unit = <fun>
# !x;;
- : int = 17
# modify x;;
- : unit = ()
# !x;;
- : int = 18
```

What about local variables?

```
# let incUpdate (n:int) = let m = ref n in (m := !m + 1);;
val incUpdate : int -> unit = <fun>
# let q = 37;;
val q : int = 37
# incUpdate q;;
- : unit = ()
# q;;
- : int = 37
```

We can write this but it is absolutely useless; what happens to m is hidden to the outside world. When `incUpdate` 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 ref 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 displayUpdate () = (Printf.printf "x is : %i," !x); (x := !x + 3);
                         (Printf.printf " Now x is : %i " !x);;
val displayUpdate : unit -> unit = <fun>
# displayUpdate ();;
x is : 24, Now x is : 27 - : unit = ()
```

The first two lines is what I typed, the next line was echoed by the interpreter. It shows that I have defined a function called `displayUpdate` with type `unit -> unit`. Since this is a function; *nothing happens until I apply it to something.* The only argument it accepts is a value of `unit` type *i.e.* `()`. The fourth line shows this and what follows is what is displayed with the function actually executes. These are the functions that you are familiar with: a sequence of commands is executed. The commands do not return a value (apart from `()`) 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 m = ref n in
  (Printf.printf "m is %i " !m);(m := !m + 1); (Printf.printf "n is %i " n);
  (Printf.printf "m is %i" !m);;
      val mash : int -> unit = <fun>
# mash 3;;
m is 3 n is 3 m is 4- : unit = ()
```

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 `done` 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 x = ref n in
  while (!x < 10) do
    (Printf.printf "x is %i\n " !x);(x := !x + 1)
  done;;
        val foo : int -> unit = <fun>
# foo 5;;
x is 5
 x is 6
 x is 7
 x is 8
```

```
 x is 9
 - : unit = ()
```

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 };;
type person = { name : string; birthday : int * int; title : string; }
# let prakash = { name = "Prakash"; birthday = (3,11);
                title = "Bane of while loops"};;
  val prakash : person =
  {name = "Prakash"; birthday = (3, 11); title = "Bane of while loops"}
# prakash.name;;
- : string = "Prakash"
```