References

So far we have examined functional programming concepts and we have shied away from the important topic of mutation, a topic familiar to those from programming languages like Java and C.

Structural Mutation and Variable Mutation

Variable Mutation

Mutation simply means "changing the value of something". For instance if we have code like the following in Python:

```
i = 0
j = i                # i = 0      j = 0
def getsum ():
    return i + j

print(getsum())
i = i + 1            # i = 1      j = 0
print(getsum())
j = j + 4            # i = 1      j = 4
print(getsum())
```

We see the value of `i` and `j` changes as the program evolves. They are still the same `i`, `j` variables, and the function `getsum` uses them. But the result of calling `getsum` depends on *when* we call `getsum`; it will use whatever value those two variables have at the time. This is known as **variable mutation**, as opposed to the *structural mutation* we will see in a little while.

Before we do that, it is important to talk about **aliasing**. Note the line `j = i`, where we have told Python to set a variable equal to another variable. Every time we do this we have to ask if we have created an "alias". Is the variable `j` from this point forward interchangeable with the variable `i`? Are they both from now on pointing to the "same value"? The answer is of course no; we can increment `i` without a corresponding change in `j`. This is in general the case with variable mutation.

Structural Mutation

There is a slightly different kind of mutation, called structural mutation, that we are all mostly familiar with from most object-oriented programming languages. In Python a simple example of it might look as follows:

```
class Counter():
   def __init__(self):
      self.i = 0

   def incr(self):
      self.i += 1
      return self.i
```

```
x = Counter()           # x.i = 0
y = x                   # literally the same "object" as x
print(x.incr())         # Increments both x and y
print(y.incr())         # Increments both x and y
```

In this example, we have an "object" that contains the value in its property `i`. And we can change the value that is stored there. At the same time, another variable `y` is bound to that same object `x`, and these two are now **aliased**: A change in the one object is reflected in the "other" (though there really isn't another object, they are the same object).

Those familiar with pointers know that pointers lie at the heart of the idea of aliasing. For example we could do the following in C:

```
#include <stdio.h>
int i = 0;
int *p1, p2, p3;

int main() {
   int *p1 = &i;
   int *p2 = p1;   // p1, p2 both point to the same memory, where i is stored
   int j = *p1;
   printf("%i␣%i␣%i␣%i\n", i, j, *p1, *p2);
   i = i + 1;
   printf("%i␣%i␣%i␣%i\n", i, j, *p1, *p2);
   *p1 += 1;
   printf("%i␣%i␣%i␣%i\n", i, j, *p1, *p2);
   return 0;
}
```

In this example we have created the two pointers that both point to the same point in memory. When we later change the value of that point in memory, that information is reflected in all pointers that still point to that same location.

This is called **structural mutation**, because we are not really changing the value of the pointers, but instead we make adjustments to the structure the pointers are pointing to.

The essential point is that these are two subtly different kinds of mutation, and it is important to identify the difference between them. Languages like OCAML allow the latter, but not the former. We will see now how.

References

Essentially references in OCAML are a bit like pointers, only safer and with limited power. If you think about it, the main things we need from pointers are the following:

1. A way to create a pointer to some new memory location.
2. A way to read the value that is stored in that location.
3. A way to change the value that is stored in that location.

Any implementation of a pointer-like mechanic must provide these three features. In a statically typed language we need some new types in addition to all that. So let us get started:

- We have a new kind of value, `ref v` where `v` is some other value. This is meant to be sort of like a pointer to the value `v`.
- We have a new type `'a ref` for a "reference to a value of type `'a`".
- We have an assignment function `r := e` which has type `'a ref * 'a -> unit`. It is evaluated by evaluating the reference `r` first, then evaluating the expression `e` to a value, then storing that value in the reference.
- We have a dereference function `!r` which has type `'a ref -> 'a`. It is evaluated by first evaluating `r` to a reference, then getting the value that is stored in there.
- Lastly, in order for these to have any practical significance, we need a way to have expressions one after the other, in a sequencing style `e1;e2`. This sequence expression evaluates `e1` first, then discards the result and evaluates `e2`.