I’ve been learning to program in OCaml for one of my courses this semester (COS 326: Functional Programming) and I haven’t been so excited by anything CS-related in a long time. Learning to program functionally is like learning to program all over again.

In this post, I’m going to touch on some of the features that make OCaml beautiful to me. In particular, I’m going to contrast these features with their Java counterparts—I think most of the observations made will generalize to the broader question of functional vis-à-vis objective-oriented/imperative, but I’m sure there are a few outliers given any specific language. Also note that I’m sticking to the functional subset of OCaml (the language incorporates some objective-oriented and imperative features, but you can get by without using them at all if you so desire).

Type definitions

OCaml makes unbelievably easy to define new types. Take, for example, the construction of a module to solve SAT problems using statements in conjunctive normal form. In Java, you would need to create a class ‘clause’, which has a private class ‘literal’, which might have a private class ‘symbol’—oh, and then you’d need to create a class CNF which has, as one of its fields, an array of clauses. I can see this boiling down to roughly 30 lines of code just for outlining your type definitions. In OCaml, we can cover all of this ground with:

type symbol = string ;;

type literal = symbol \* bool ;;

type clause = literal list ;;

type cnf = clause list ;;

Most of what’s happening here can be read out-loud: a ‘symbol’ is just a string; a literal is just a ‘symbol’ and a Boolean (i.e., negated or not); a clause is just a list of literals; and a CNF is just a list of clauses. Brilliant. (See here for some work I did on OCaml SAT solvers.)

Better yet, **we can define a binary tree in one line**:

type tree = Leaf | Node of int \* tree \* tree ;;

This is saying that a tree is either a leaf, or a node with some value and two subtrees. That is the entire definition.

Generics

We can make our tree generic with:

type ‘a tree = Leaf | Node of ‘a \* ‘a tree \* ‘a tree ;;

Think about how much time would be spent in Java handling the Integer or Object wrapper classes—all of that is ignore dwith the ‘a semantic in OCaml, which lets you fill in the type later (the type checker will still detect type mismatches! E.g., if you attempt to add a char tree as a subtree of an int tree, you will get a compile-time error).

The handling of generic types in OCaml really couldn’t be easier.

Option types

OCaml also provides a framework for handling what in Java might be called the Null case through its option types. By specificying a variable as type ‘a option (e.g., int option), that variable can then either take on the form None or Some ‘a.

For example, if x is of type int option, then x can either be None or Some int (e.g., Some 12, Some -1, Some 0). This becomes very useful, very quickly: think about functions in which you may want to return nothing (due to some bad input) but would prefer not to throw an exception. With the option type, we’re able to avoid all the extreme dangers that you see so often in C regarding null pointers and the general disruption that null values can cause.

The Thought Process

An example that really instilled in me an appreciation for OCaml came when attempting to check whether two lists are of equal length. The function looks like this:

let equal\_length l1 l2 : bool =

match (l1, l2) with

([], []) -> true

| ([], \_) -> false

| (\_, []) -> false

| (\_::tl1, \_::tl2) -> equal\_length tl1 tl2 ;;

The semantics can be ignored, but the basic premise is that we continue to pull one item at a time off the two lists until they are either both empty (in which case the lengths must be equal) or one is empty and the other is not (in which case the lengths must not be equal).

To me, this was a great example of how OCaml trains programmers to think both deeply and recursively: here, we established an invariant (“we will take one item at a time off both lists”), and from that we were able to reach conclusions given the structure of the resulting lists.

OCaml is constantly training you to think recursively, like with the above. I can’t recall a single instance in which I’ve used a for-loop; instead, OCaml provides you with an awesome framework for exercising induction—indeed, in most cases, you’ll recur on the naturally recursive data structures that OCaml provides you with. For example, consider summing a tree of integers using the Tree definition above.

let sum tree =

match tree with

Leaf -> 0

| (value, left, right) -> value + sum left + sum right ;;

Here, we’re taking the sum of a node and adding it to the sum of all nodes in each of the node’s two subtrees. Recursion is awesome.

Conclusion

This post ignores a lot of the best or most powerful features of OCaml, but it’s dragged on a little long, so I’ll stop here. For those interested, check out: proving program correctness (according to my professor: conquerable by college students in OCaml, but “still the subject of current research” in Java), currying functions, and pattern matching.