CSE341, Fall 2011, Lecture 11 Summary

Standard Disclaimer: This lecture summary is not necessarily a complete substitute for attending class, reading the associated code, etc. It is designed to be a useful resource for students who attended class and are later reviewing the material.

To conclude our study of function closures, we digress from ML to show similar programming patterns in Java (using generics and interfaces) and C (using function pointers taking explicit environment arguments). (While we won’t test you on this material, it may nonetheless help you understand closures by seeing similar ideas in other settings, and it should help you see how central ideas in one language can influence how you might approach problems in other languages.)

An ML Example

Consider this ML code where we define our polymorphic linked-list type constructor and three polymorphic functions (two higher-order) over that type. We will investigate a couple ways we could write similar code in Java or C, which will can help us better understand similarities between closures and objects (for Java) and how environments can be made explicit (for C). In ML, there is no reason to define our own type constructor since ‘a list is already written, but doing so will help us compare to the Java and C versions.

datatype ‘a mylist = Cons of ‘a * (‘a mylist) | Empty

fun map f xs =
    case xs of
    | Empty => Empty
    | Cons(x,xs) => Cons(f x, map f xs)

fun filter f xs =
    case xs of
    | Empty => Empty
    | Cons(x,xs) => if f x then Cons(x,filter f xs) else filter f xs

fun length xs =
    case xs of
    | Empty => 0
    | Cons(_,xs) => 1 + length xs

Using this library, here are two client functions. (The latter is not particularly efficient, but shows a simple use of length and filter.)

val doubleAll = map (fn x => x * 2)
fun countNs (xs, n : int) = length (filter (fn x => x=n) xs)

Closures via Java Interfaces

A future version of Java is likely to have convenient support for closures just like most other mainstream object-oriented languages now do (C#, Scala, Ruby, ...), but it is worth considering how we might write similar code in Java today. While we don’t have first-class functions, currying, or type inference, we do have generics (Java didn’t used to) and we can define interfaces with one method, which we can use like function types. Without further ado, here is a Java analogue of the code, followed by a brief discussion of features you may not have seen before and other ways we could have written the code:
interface Func<B,A> {
    B m(A x);
}
interface Pred<A> {
    boolean m(A x);
}
class List<T> {
    T head;
    List<T> tail;
    List(T x, List<T> xs) {
        head = x;
        tail = xs;
    }
    static <A,B> List<B> map(Func<B,A> f, List<A> xs) {
        if(xs==null)
            return null;
        return new List<B>(f.m(xs.head), map(f,xs.tail));
    }
    static <A> List<A> filter(Pred<A> f, List<A> xs) {
        if(xs==null)
            return null;
        if(f.m(xs.head))
            return new List<A>(xs.head, filter(f,xs.tail));
        return filter(f,xs.tail);
    }
    static <A> int length(List<A> xs) {
        int ans = 0;
        while(xs != null) {
            ++ans;
            xs = xs.tail;
        }
        return ans;
    }
}
class ExampleClients {
    static List<Integer> doubleAll(List<Integer> xs) {
        return List.map((new Func<Integer,Integer>() {
                        public Integer m(Integer x) { return x * 2; }
                    },
                    xs);
    }
    static int countNs(List<Integer> xs, final int n) {
        return List.length(List.filter((new Pred<Integer>() {
                        public boolean m(Integer x) { return x==n; }
                    })),
                    xs);
    }
}

This code uses several interesting techniques and features:
• In place of the (inferred) function types `a -> 'b for map and 'a -> bool for filter, we have generic interfaces with one method. A class implementing one of these interfaces can have fields of any types it needs, which will serve the role of a closure’s environment.

• The generic class List serves the role of the datatype binding. The constructor initializes the head and tail fields as expected, using the standard Java convention of null for the empty list.

• Static methods in Java can be generic provided the type variables are explicitly mentioned to the left of the return type. Other than that and syntax, the map and filter implementations are similar to their ML counterparts, using the one method in the Func or Pred interface as the function passed as an argument. For length, we could use recursion, but choose instead to follow Java’s preference for loops.

• If you have never seen anonymous inner classes, then the methods doubleAll and countNs will look quite odd. Somewhat like anonymous functions, this language feature lets us crate an object that implements an interface without giving a name to that object’s class. Instead, we use new with the interface being implemented (instantiating the type variables appropriately) and then provide definitions for the methods. As an inner class, this definition can use fields of the enclosing object or final local variables and parameters of the enclosing method, gaining much of the convenience of a closure’s environment with more cumbersome syntax. (Anonymous inner classes were added to Java to support callbacks and similar idioms.)

There are many different ways we could have written the Java code. Of particular interest:

• Tail recursion is not as efficient as loops in implementations of Java, so it is reasonable to prefer loop-based implementations of map and filter. Doing so without reversing an intermediate list is more intricate than you might think (you need to keep a pointer to the previous element, with special code for the first element), which is why this sort of program is often asked at programming interviews. The recursive version is easy to understand, but would be unwise for very long lists.

• A more object-oriented approach would be to make map, filter, and length instance methods instead of static methods. The method signatures would change to:

        <B> List<B> map(Func<B,T> f) {...}
        List<T> filter(Pred<T> f) {...}
        int length() {...}

The disadvantage of this approach is that we have to add special cases in any use of these methods if the client may have an empty list. The reason is empty lists are represented as null and using null as the receiver of a call raises a NullPointerException. So methods doubleAll and countNs would have to check their arguments for null to avoid such exceptions.

• Another more object-oriented approach would be to not use null for empty lists. Instead we would have an abstract list class with two subclasses, one for empty lists and one for nonempty lists. This approach is a much more faithful object-oriented approach to datatypes with multiple constructors, and using it makes the previous suggestion of instance methods work out without special cases. It does seem more complicated and longer to programmers accustomed to using null.

• Anonymous inner classes are just a convenience. We could instead define “normal” classes that implement Func<Integer,Integer> and Pred<Integer> and create instances to pass to map and filter. For the countNs example, our class would have an int field for holding n and we would pass the value for this field to the constructor of the class, which would initialize the field.
Explicit Environments with C Function Pointers

C does have functions, but they are not closures. If you pass a pointer to a function, it is only a code pointer. As we have studied, if a function argument can use only its arguments, higher-order functions are much less useful. So what can we do in a language like C? We can change the higher-order functions as follows:

- Take the environment explicitly as another argument.
- Have the function-argument also take an environment.
- When calling the function-argument, pass it the environment.

So instead of a higher-order function looking something like this:

```c
int f(int (*g)(int), list_t xs) { ... g(xs->head) ... }
```

we would have it look like this:

```c
int f(int (*g)(void*,int), void* env, list_t xs) { ... g(env,xs->head) ... }
```

We use `void*` because we want `f` to work with functions that use environments of different types, so there is no good choice. Clients will have to cast to and from `void*` from other compatible types. We don’t discuss those details here.

While the C code has a lot of other details, this use of explicit environments in the definitions and uses of `map` and `filter` is the key difference from the versions in other languages:

```c
#include <stdlib.h>
#include <stdint.h>
#include <stdbool.h>

typedef struct List list_t;
struct List {
    void * head;
    list_t * tail;
};
list_t * makelist (void * x, list_t * xs) {
    list_t * ans = (list_t *)malloc(sizeof(list_t));
    ans->head = x;
    ans->tail = xs;
    return ans;
}
list_t * map(void* (*f)(void*,void*), void* env, list_t * xs) {
    if(xs==NULL)
        return NULL;
    return makelist(f(env,xs->head), map(f,env,xs->tail));
}
list_t * filter(bool (*f)(void*,void*), void* env, list_t * xs) {
    if(xs==NULL)
        return NULL;
    if(f(env,xs->head))
        return makelist(xs->head, filter(f,env,xs->tail));
```
As in Java, using recursion instead of loops is much simpler but likely less efficient. Another alternative would be to define structs that put the code and environment together in one value, but our approach of using is an extra void* argument to every higher-order function is more common in C code.