CSci 4223 Lecture 12 March 4, 2013

Topics: Lexical scope, dynamic scope, closures

---

1 Lexical Scope

SML, and nearly all modern languages, follow the **Rule of Lexical Scope:** *the body of a function is evaluated in the old dynamic environment that existed at the time the function was **defined**, not the current environment when the function is **called**.*

Here is a silly example to demonstrate the consequences of the rule:

```
val x = 1
fun f y = x + y
val x = 2
val y = 3
val z = f (x+y)
```

In this example, `f` is bound to a function that takes an argument `y`. The body of `f` uses `x`. According to the rule given above, since the environment at `f`'s definition maps `x` to 1, function `f` must **always** return the result of evaluating `1+y`. On the final line of the example, the original binding of `x` (to `1`) has been shadowed by a new binding (to `2`). Here is how the evaluation of that line proceeds:

- Look up `f` in the current dynamic environment to get the previously described function.
- Evaluate the argument `x+y` in the current dynamic environment by looking up `x` and `y`, producing `5`.
- Call the function with argument `5`.
- Evaluate the body `x+y` in the *old* dynamic environment where `x` maps to 1 (i.e., the dynamic environment that existed at the point in time when `f` was bound), extended with a new binding of `y` to `5`.
- Since `1+5` is `6`, the result is `6`.

The alternative to lexical scope is the **Rule of Dynamic Scope**: *the body of a function is evaluated in the current dynamic environment at the time the function is **called**, not the old dynamic environment that existed at the time the function was **defined**.* Dynamic scope would produce `7` as the final result in the above example:

- Look up `f` in the current dynamic environment to get the previously described function.
- Evaluate the argument `x+y` in the current dynamic environment by looking up `x` and `y`, producing `5`.
- Call the function with argument `5`.
- Evaluate the body `x+y` in the *current* dynamic environment where `x` maps to 2, extended with a new binding of `y` to `5`.
- Since `2+5` is `7`, the result is `7`.

Once upon a time, static and dynamic scoping might have been considered equally valid alternatives for programming languages. But decades of experience have taught us the static scoping is actually superior. To see why, let's first examine function values more closely.

2 Closures

In the definition of `f` above (repeated here),

```
fun f y = x + y
```

variable `x` is not a parameter of `f`. We say that `x` is a *free variable* of `f`, because it is not *bound* by `f`. So how does the run-time determine what value `x` has? The answer has to do with what functions values are.

So far, we have said that functions are values, but we have not been precise about what those values exactly are. It turns out that a function value has two parts: the *code* for the function, and the *environment* that was current when the function was defined. That environment is what determines the values of free variables. You can think of these two parts as being a "pair," but it is not an ML pair—just something with two parts. You cannot access the parts of the "pair" separately; all you can do is call the function. When called, both parts get used—the code determines what computation occurs, and the environment determines what the values of free variables are.

This "pair" is called a *function closure* or just *closure*. The reason for the name is that the closure's environment "closes" off the free variables in the code of the function. In the example above, binding `fun f y = x + y` actually binds `f` to a closure. The code part of that closure is the function `fn y => x + y`. The environment part maps free variable `x` to 1. Therefore, any call to this closure will return the result of evaluating `1+y`.

Example 1.

```
val x = 1
fun f y =
  let
    val x = y+1
  in
    fn z => x+y+z
  end
val x = 3
val g = f 4
val y = 5
val z = g 6
```

Here, `f` is bound to a closure in which the environment part maps `x` to `1`. So in the evaluation of `f 4`, expression `let val x = y+1 in fn z => x+y+z end` is evaluated in an environment where `x` maps to `1` and `y` maps to `4`. But the `let` expression then shadows `x`, such that it maps to `5`. So `fn z => x+y+z` is evaluated in an environment where `x` maps to `5` and `y` maps to 4. Evaluating `fn z => x+y+z` therefore creates a closure. The code part of that closure is just `fn z => x+y+z`. The environment part is the dynamic environment that exists at the time the closure is created—that is, `x` maps to `5` and `y` maps to 4. So `g` is bound to a closure that, when called, will always add `9` to its argument, no matter what the environment is at any call-site. Therefore, in the last line of the example, `z` will be bound to `15`.

Example 2:

```
fun f g =
  let
    val x = 3
  in
    g 2
  end
val x = 4
fun h y = x + y
val z = f h
```

Here, `f` is bound to a closure that takes another function `g` as an argument and returns the result of `g 2`. The closure bound to `h` *always* adds `4` to its argument because `h` is defined in an environment where `x` maps to `4`. So in the last line, `z` will be bound to `6`. The binding `val x = 3` is totally irrelevant: `g 2` is evaluated by looking up `g` to get the closure that was passed in and then using that closure with *its* environment part (in which `x` maps to `4`).

Example 3.

```
fun filter (f,xs) =
  case xs of
    [] => []
  | x::xs' => if f x then x::(filter(f,xs')) else filter(f,xs')

fun allGreaterThanSeven xs = filter (fn x => x > 7, xs)
fun allGreaterThan (xs,n) = filter (fn x => x > n, xs)
```

Here, `allGreaterThanSeven` is "old news"—we pass in a function that removes from the result any numbers 7 or less in a list. But it is much more likely that you want a function like `allGreatherThan` that takes the "limit" as a parameter `n` and uses the function `fn x => x > n`. Notice this requires a closure and lexical scope. When the implementation of `filter` calls this function, the run-time needs to look up `n` in the environment where `fn x => x > n` was defined.

Example 4.

```
fun allShorterThan1 (xs,s) = filter (fn x => String.size x < String.size s, xs)

fun allShorterThan2 (xs,s) =
  let
    val i = String.size s
  in
    filter(fn x => String.size x < i, xs)
  end
```

Both these functions take a list of strings `xs` and a string `s` and return a list containing only the strings in `xs` that are shorter than `s`. And they both use closures to look up `s` or `i` when the anonymous functions get called. The second one is more complicated but a bit more efficient: it "precomputes" `String.size s` and binds it to a variable `i` available to the function `fn x => String.size x < i`. Whereas, the first one recomputes `String.size s` once per element in `xs` (because `filter` calls its function argument this many times and the body evaluates `String.size s` each time).

3 Lexical vs. dynamic scope

Dynamic scope leads to serious problems!

- Suppose, in Example 1, the body of `f` were changed to `let val q = y+1 in fn z => q+y+z`. Under lexical scope this is fine: we can always change the name of a local variable and its uses without it affecting anything. But under dynamic scope, now the call to `g 6` will make no sense: we will try to look up `q`, but there is no `q` in the environment at the call-site.
- Consider again the original version of Example 1, but now change the line `val x = 3` to `val x = "hi"`. Under lexical scope, this is again fine: that binding is never actually used. But under dynamic scope, the call to `g 6` will look up `x`, get a string, and try to add an integer to it. That would be a type error.
- In Example 2, the body of `f` is unstylish: there is a local binding that is never used. Under lexical scope we can remove it, changing the body to `g 2`. This won't have any impact on the rest of the program. But under dynamic scope, it would change the final value of `z` to `5`, because `x` would be bound to `3` instead of `4` during the evaluation of `h`'s body.

For "regular" variables in programs, lexical scope is the best choice, without question. It enables programmers to better reason about what a function does—for example, under lexical scope we are guaranteed that any use of the closure bound to `h` will add `4` to its argument, regardless of how other functions like `g` are implemented and what variable names they use. This is a key separation-of-concerns that only lexical scope provides.

There are limited cases where dynamic scope does make sense. But they are sufficiently rare that few modern languages support it or provide it as the default. In fact, many languages that began life offering dynamic scope now either offer a choice between both scoping rules (e.g., Perl) or have developed variants that are only lexically scoped (e.g., LISP and Scheme).

One known good use of dynamic scope is exception handling. When an exception is raised, evaluation has to "look up" which handle expression should be evaluated. This "look up" is done using the dynamic call stack, with no regard for the lexical structure of the program.

4 Closures with Java

Java 8 will have support for closures, just like most other mainstream object-oriented languages now do (C#, Scala, Ruby, ...). But the details aren't finalized yet, so let's consider how we might write closure-like code in Java today. Although Java doesn't currently have first-class functions, currying, or type inference, it does have *generics*, which we can use to code up something like closures.

Let's start with this ML code:[1]

```
datatype 'a mylist = Cons of 'a * ('a mylist) | Nil

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

fun map f xs =
    case xs of
        Nil => Nil
```

[1] In ML, there is no reason to define `'a mylist` since `'a list` is already provided, but doing so will help us compare ML to Java.

```
      | Cons(x,xs) => Cons(f x, map f xs)

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

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

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<A,B> {
    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<A,B> 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 Java features:

- Generic class `List` is used instead of an ML type constructor. The `List` class constructor plays the role of the `Cons` datatype constructor. As is idiomatic in Java, `null` is used for the empty list, whereas the ML code uses the `Nil` datatype constructor.
- Generic interface `Func` is used instead of first-class functions—because Java doesn't have first-class functions. An instance of `Func` represents a function of ML type `'a -> 'b`. An instance of `Pred` represents a function of ML type `'a -> bool`. The function "inside" an instance is invoked with method `m`; that method provides the equivalent of a closure's code part. The fields of a class implementing one of these interfaces could be used to provide the equivalent of a closure's environment part.
- The `map` and `filter` implementations are similar to their ML counterparts. The use the `m` method passed to them as an argument (inside an object of type `Func` or `Pred`) as the equivalent of the first-class function that would instead be passed in ML. Note that generic, static methods in Java require the type variables to be explicitly mentioned to the left of the return type.
- Methods `doubleAll` and `countNs` use *anonymous inner classes*, which you've probably encountered before when programming with Java's Swing GUI library. This language feature lets us create an object that implements an interface without giving a name to that object's class. (In that respect, it's similar to anonymous functions in ML.) 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.

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

- The Java code uses recursion to implement `map` and `filter`. Tail recursion is not as efficient as loops in Java, so for efficiency's sake, we should probably rewrite `map` and `filter` to use loops. However, doing so—without needing to reverse an intermediate list—is more intricate than you might think. That's why sort of program is often given as a challenge problem at programming-job 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<T,B> f) {...}
  List<T> filter(Pred<T> f) {...}
  int length() {...}
  ```

  The disadvantage of this approach is that we would have to add code at any call-site `lst.map(...)` or `lst.filter(...)` of these methods to check whether `lst` is the empty list—that is, `null`. Otherwise, trying to invoke an instance method could raise a `NullPointerException`. Likewise, methods `doubleAll` and `countNs` would have to check their arguments for `null` to avoid such exceptions.

- An even more object-oriented (though less idiomatic in Java) 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.

The moral of the story is that we can code-up function closures and higher-order functions in Java as it is today, though it's a little more difficult than coding up the same thing in ML.

5 Closures with C

C does have first-class functions. To make a function `f` take another function `g` as an argument, you can write this code:

```
int f(int (*g)(int)) { ... g(17) ... }
```

Function `g` takes an `int` as an argument and returns an `int`, which is what the perhaps strange-looking syntax indicates.

However, C's first-class functions are **not** closures. If you pass a pointer to a function, it is only a code pointer. There is no environment part, as there was in ML closures. That's unfortunate, because a lot of the convenience of higher-order functions in ML came from the fact that they could use lexically scoped, free variables.

However, we can code-up closures in C. The way to "give a function `f` a closure" is to modify `f` to explicitly take its environment as an argument. That environment will need to be supplied by any functions that call `f`, so those functions will also need to take the environment as an argument.

So instead of a function being implemented like this:

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

it would instead be implemented like this:

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

We use `void*` as the type of the environment, because there isn't any better choice. Clients will have to cast to and from `void*` when passing and using environments. We don't discuss those details here.

Here's the full C version of our running example:

```
#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));
  return filter(f,env,xs->tail);
}
int length(list_t* xs) {
  int ans = 0;
  while(xs != NULL) {
    ++ans;
    xs = xs->tail;
  }
  return ans;
}
void* doubleInt(void* ignore, void* i) { // type casts to match what map expects
  return (void*)(((intptr_t)i)*2);
}
list_t * doubleAll(list_t * xs) { // assumes list holds intptr_t fields
  return map(doubleInt, NULL, xs);
}
bool isN(void* n, void* i) { // type casts to match what filter expects
  return ((intptr_t)n)==((intptr_t)i);
}
int countNs(list_t * xs, intptr_t n) { // assumes list hold intptr_t fields
  return length(filter(isN, (void*)n, xs));
}
```

There are a lot of details we could discuss about this code, but we'll limit ourselves to just two:

- As in Java, using recursion instead of loops is much simpler but likely less efficient.
- Instead of modifying functions to take environments as arguments, another alternative would be to define structs that put the code and environment together in one value. However, our approach (an extra `void*` argument to every higher-order function) is probably more idiomatic in C.