CSE 341, Winter 2008, Lecture 9 Summary

Standard Disclaimer: These comments may prove useful, but certainly are not a complete summary of all the important stuff we did in class. They may make little sense if you missed class, but hopefully will help you organize and process what you have learned.

We continue describing programming idioms enabled by function closures.

Implementing an Abstract Data Type

The key to an abstract data type is requiring clients to use it via a collection of functions rather than directly accessing its private state. Thanks to this abstraction, we can later change how the data type is implemented without changing how it behaves for clients. In an object-oriented language, you might implement an abstract data type be defining a class with all private fields (inaccessible to clients) and some public methods (the interface with clients). We can do the same thing in ML with a record of closures; the variables that the closures use from the environment correspond to the private fields. Admittedly, this programming idiom can appear very fancy/clever/subtle, but it suggests (correctly) that functional programming and object-oriented programming are more similar than they might first appear (a topic we will revisit later in the course; there are important differences).

As an example, consider an implementation of a set of integers that supports creating a new bigger set and seeing if an integer is in a set. Our sets are mutation-free in the sense that adding an integer to a set produces a new, different set. In ML, we could define a type that describes this interface:

```
datatype set = S of { add : int -> set, member : int -> bool }
```

Roughly speaking, a set is a record with two fields, each of which holds a function. It would be simpler to write:

```
type set = { add : int -> set, member : int -> bool }
```

but this does not work in ML because `type` bindings cannot be recursive. So we have to deal with the mild inconvenience of having a constructor `S` around our record of functions defining a set. Notice we are not using any new types or features; we simply have a type describing a record with fields named `add` and `member`, each of which holds a function.

Once we have an empty set, we can use its `add` field to create a one-element set, and then use that set's `add` field to create a two-element set and so on. So the only other thing our interface needs is a binding like this:

```
val empty_set = ... : set
```

Before implementing this interface, let's see how a client might use it:

```
fun use_a_set () =
    let val S s1 = empty_set
        val S s2 = (#add s1) 34
        val S s3 = (#add s2) 19
    in
        if (#member s3) 42
        then 99
        else if (#member s3) 19
        then 17
        else 0
    end
```

Again we are using no new features. `#add s1` is reading a record field, which in this case produces a function that we can then call with 34. If we were in Java, we might write `s1.add(34)` to do something similar. The `val` bindings use pattern-matching to "get rid of" the `S` constructors on values of type `set`.

There are many ways we could define `empty_set`; they will all use the technique of using a closure to "remember" what elements a set has. Here is one way:

```
val empty_set =
    let fun exists(j,lst) = (* could use currying and/or fold to be fancier *)
            case lst of
                [] => false
              | hd::tl => j=hd orelse exists(j,tl)
        fun make_set lst = (* lst is a "private field" *)
            S { add    = fn i => make_set (i::lst),
                member = fn i => exists (i,lst) }
    in
        make_set []
    end
```

The helper function `exists` just sees if a list has an element; we could also just use `List.exists` in the standard library. All the fanciness is in `make_set` and `empty_set` is just the record returned by `make_set []`. What `make_set` returns is a value of type `set`. It is essentially a record with two functions. The closures produced from `fn i => make_set (i::lst)` and `fn i => exists (i,lst)` are values that *when called* use `lst` — which is the "private field" we need to produce either a `bool` (for `member`) or a new set (for `add`).

Callbacks

Another common idiom is to implement a library that detects when "events" occur and informs clients that have previously "registered" their interest in hearing about events. Clients can register their interest by providing a "callback" — a function that gets called when the event occurs. Examples of events for which you might want this sort of library include things like users moving the mouse or pressing a key. Data arriving from a network interface is another example.

The purpose of these libraries is to allow multiple clients to register callbacks. The library implementer has no idea what clients need to compute when an event occurs, and the clients may need "extra data" to do the computation. So the library implementor should not restrict what "extra data" each client uses. A closure is ideal for this because a function's type `t1 -> t2` doesn't specify the types of any other variables a closure uses, so we can put the "extra data" in the closure's environment.

If you have used "event listeners" in Java's Swing library, then you have used this idiom in an object-oriented setting. In Java, you get "extra data" by defining a subclass with additional fields. This can take an awful lot of keystrokes for a simple listener, which is a (the?) main reason the Java language added anonymous inner classes (which you do not need to know about for this course), which are closer to the convenience of closures.

To see an example in ML, we will finally introduce ML's support for mutation. Mutation is okay in some settings. In this case, we really do want registering a callback to "change the state of the world" — when an event occurs, there are now more callbacks to invoke. In ML, most things really cannot be mutated. Instead you must create a *reference*, which is a container whose contents can be changed. You create a new reference with the expression `ref e` (the initial contents are the result of evaluating `e`). You get a reference `r`'s current contents with `!r` (not to be confused with negation in Java or C), and you change `r`'s contents with `r := e`. The type of a reference that contains values of type `t` is written `t ref`.

Our example will use the idea that callbacks should be called when a key on the keyboard is pressed. We will pass the callbacks an `int` that encodes which key it was. Our interface just needs a way to register callbacks. (In a real library, you might also want a way to unregister them.)

```
val onKeyEvent : (int -> unit) -> unit
```

Clients will pass a `int -> unit` that, when called later with an `int` will do whatever they want. To implement this function, we just use a reference that holds a list of the callbacks. Then when an event actually occurs, we assume the function `on_event` is called and it calls each callback in the list:

```
val cbs : (int -> unit) list ref = ref []
fun onKeyEvent f = cbs := f::(!cbs)
fun on_event i =
```

```
  let fun loop l =
       case l of
         [] => []
       | f::tl => f i; loop tl
   in loop (!cbs) end
```

Most importantly, the type of `onKeyEvent` places no restriction on what extra data a callback can access when it is called. Here are three different clients (calls to `onKeyEvent` that use different variables in functions in their environment:

```
onKeyEvent (fn i => write_to_log(Int.toString i
                                 ^ " got pressed\n"));

val f4_key = 75; (* no idea what it really is *)
onKeyEvent (fn i => if i=f4_key then minimizeWindow() else ());

fun prohibit_keys lst =
  onKeyEvent (fn i => if (List.exists (fn j => j=i) lst)
                      then exitProgram()
                      else ());
prohibit_keys [13, 42, 99];
```

We describe the `List.exists` library function below.

Partial application ("currying")

The final idiom we consider is very convenient, especially when defining and using iterators over data structures (see last lecture). We have already seen that in ML every function takes exactly one argument, so you have to use an idiom to get the effect of multiple arguments. Previously we have done this by passing a tuple as the one argument, so each part of the tuple is conceptually one of the multiple arguments. Another more clever and often more convenient way is to have a function take the first conceptual argument and return another function that takes the second conceptual argument and so on.

This technique is called "currying" because someone named Curry was a researcher who studied the idea.

Here is an example of a "3-argument" function that uses currying:

```
val inorder3 = fn x => fn y => fn z =>
                 z >= y andalso y >= x
```

If we call `inorder3 4` we will get a closure that has `x` in its environment. If we then call this closure with 5, we get a closures that has `x` and `y` in its environment. If we then call this closure with 6, we will get `true` because 6 is greater than 5 and 5 is greater than 4. That is just how closures work.

So `((inorder3 4) 5) 6` computes exactly what we want and feels pretty close to calling `inorder3` with 3 arguments. Even better, the parentheses are optional, so we can write exactly the same thing as `inorder3 4 5 6`, which is actually fewer non-space characters than our old tuple approach where we would have:

```
fun inorder3 (x,y,z) = z >= y andalso y >= x
val someClient = inorder3(4,5,6)
```

Moreover, even though we might expect most clients of our curried `inorder3` to provide all 3 conceptual arguments, they might provide fewer and use the resulting closure later. This is called "partial application" because we are providing a subset (more precisely, a prefix) of the conceptual argument. As a silly example, `inorder3 0 0` returns a function that returns `true` if its argument is nonnegative.

Currying is particularly convenient for creating similar functions with iterators. For example, here is a curried version of a fold function for lists: