CSE341, Fall 2011, Lecture 3 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.

This lecture has three topics:

- Let-expressions, an absolutely crucial feature that allows for local variables in a very simple, general and flexible way. It is crucial for style and for efficiency.
- Options, which are a way to build data that has 0 or 1 items. We could use 0-element and 1-element lists instead, but using options is better style because it makes clear that the number of items must be 0 or 1.
- Benefits of not being able to mutate (i.e., assign to) variables and parts of data structures.

Let expressions

A let-expression lets us have local variables. In fact, it lets us have local *bindings* of any sort, including function bindings. Because it is a kind of expression, it can appear anywhere an expression can.

Syntactically, a let-expression is:

```
let b1 b2 ... bn in e end
```

where each `bi` is a binding and `e` is an expression.

The type-checking and semantics of a let-expression is much like the semantics of the top-level bindings in our ML program. We evaluate each binding in turn, creating a larger environment for the subsequent bindings. So we can use all the earlier bindings for the later ones, and we can use them all for `e`. We call the *scope* of a binding "where it can be used," so the scope of a binding in a let-expression is the later bindings in that let-expression and the "body" of the let-expression (the `e`). The value `e` evaluates to is the value for the entire let-expression, and, unsurprisingly, the type of `e` is the type for the entire let-expression.

For example, this expression evaluates to 7; notice how one inner binding for `x` *shadows* an outer one.

```
let val x = 1
in
    (let val x = 2 in x+1 end) + (let val y = x+2 in y+1 end)
end
```

Also notice how let-expressions are expressions so they can appear as a subexpression in an addition (though this example is silly and bad style because it is hard to read).

Let-expressions can bind functions too, since functions are just another kind of binding. If a helper function is needed by only one other function and is unlikely to be useful elsewhere, it is good style to bind it locally. For example, here we use a local helper function to help produce the list `[1,2,...,x]`:

```
fun countup_from1 (x:int) =
    let fun count (from:int, to:int) =
            if from=to
            then to::[]
```

```
            else from :: count(from+1,to)
    in
        count(1,x)
    end
```

However, we can do better. When we evaluate a call to `count`, we evaluate `count`'s body in a dynamic environment that is the environment where `count` was defined, extended with bindings for `count`'s arguments. The code above does not really utilize this: `count`'s body uses only `from`, `to`, and `count` (for recursion). It could also use `x`, since that is in the environment when `count` is defined. Then we do not need `to` at all, since in the code above it always has the same value as `x`. So this is better style:

```
fun countup_from1_better (x:int) =
    let fun count (from:int) =
            if from=x
            then x::[]
            else from :: count(from+1)
    in
        count 1
    end
```

This technique — define a local function that uses other variables in scope — is a hugely common and convenient thing to do in functional programming. It is a shame that many non-functional languages have little or no support for doing something like it.

Local variables are often good style for keeping code readable. They can be much more important than that when they bind to the *results of* potentially expensive computations. For example, consider this code that does not use let-expressions:

```
fun bad_max (lst : int list) =
    if null lst
    then 0
    else if null (tl lst)
    then hd lst
    else if hd lst > bad_max(tl lst)
    then hd lst
    else bad_max(tl lst)
```

If you call `bad_max` with `countup_from1 30`, it will make approximately 2^{30} (over one billion) recursive calls to itself. The reason is an "exponential blowup" — the code calls `bad_max(tl lst)` twice and each of those calls call `bad_max` two more times (so four total) and so on. This sort of programming "error" can be difficult to detect because it can depend on your test data (if the list counts down, the algorithm makes only 30 recursive calls instead of 2^{30}).

We can use let-expressions to avoid repeated computations. This version computes the max of the tail of the list once and stores the resulting value in `tl_ans`.

```
fun good_max (lst : int list) =
    if null lst
    then 0
    else if null (tl lst)
    then hd lst
    else
```

```
        (* for style, could also use a let-binding for hd lst *)
        let val tl_ans = good_max(tl lst)
        in
            if hd lst > tl_ans
            then hd lst
            else tl_ans
        end
```

Options

The previous example does not properly handle the empty list — it returns 0. This is bad style because 0 is really not the maximum value of 0 numbers. There is no good answer, but we should deal with this case reasonably. One possibility is to raise an exception; you can learn about SML exceptions on your own if you are interested. Instead, let's change the return type to either return the maximum number or indicate the input list was empty so there is no maximum. Given the constructs we have, we could "code this up" by return an `int list`, using `[]` if the input was the empty list and a list with one integer (the maximum) if the input list was not empty.

While that works, lists are "overkill" — we will always return a list with 0 or 1 elements. So a list is not really a precise description of what we are returning. The ML library has "options" which are a precise description: an option value has either 0 or 1 thing: `NONE` is an option value "carrying nothing" whereas `SOME e` evaluates `e` to a value `v` and becomes the option carrying the one value `v`. The type of `NONE` is `'a option` and the type of `SOME e` is `t option` if `e` has type `t`.

Given a value, how do you use it? Just like we have `null` to see if a list is empty, we have `isSome` which evaluates to `false` if its argument is `NONE`. Just like we have `hd` and `tl` to get parts of lists (raising an exception for the empty list), we have `valOf` to get the value carried by `SOME` (raising an exception for `NONE`).

Using options, here is a better version with return type `int option`:

```
fun better_max (lst : int list) =
    if null lst
    then NONE
    else
        let val tl_ans = better_max(tl lst)
        in if isSome tl_ans andalso valOf tl_ans > hd lst
           then tl_ans
           else SOME (hd lst)
        end
```

The version above works just fine and is a reasonable recursive function because it does not repeat any potentially expensive computations. But it is both awkward and a little inefficient to have each recursive call except the last one create an option with `SOME` just to have its caller access the value underneath. Here is an alternative approach where we use a local helper function for non-empty lists and then just have the outer function return an option. Notice the helper function would raise an exception if called with `[]`, but since it is defined locally, we can be sure that will never happen.

```
fun better_max2 (lst : int list) =
    if null lst
    then NONE
    else let (* fine to assume argument nonempty because it is local *)
            fun max_nonempty (lst : int list) =
                if null (tl lst) (* lst better not be [] *)
```

```
            then hd lst
            else let val tl_ans = max_nonempty(tl lst)
                 in
                     if hd lst > tl_ans
                     then hd lst
                     else tl_ans
                 end
    in
        SOME (max_nonempty lst)
    end
```

Lack of Mutation and Benefits Thereof:

In ML, there is no way to *change* the contents of a binding, a tuple, or a list. If `x` maps to some value like the list of pairs `[(3,4),(7,9)]` in some environment, then `x` will forever map to that list in that environment. There is no assignment statement that changes `x` to map to a different list. (You can introduce a new binding that shadows `x`, but that will not affect any code that looks up the "original" `x` in an environment.) There is no assignment statement that lets you change the head or tail of a list. And there is no assignment statement that lets you change the contents of a tuple. So we have constructs for building compound data and accessing the pieces, but no constructs for *mutating* the data we have built.

This is a really powerful feature! That may surprise you: how can a language *not* having something be a feature? Because if there is no such feature, then when you are writing *your code* you can rely on *no other code* dong something that would make your code wrong, incomplete, or difficult to use. Having *immutable data* is probably the most important "non-feature" a language can have, and it is one of the main contributions of functional programming.

While there are various advantages to immutable data, here we will focus on a big one: it makes sharing and aliasing irrelevant. Let's re-consider two examples from the previous lecture before picking on Java (and every other language where mutable data is the norm and assignment statements run rampant).

```
fun sort_pair (pr : int*int) =
    if (#1 pr) > (#2 pr)
    then pr
    else ((#2 pr),(#1 pr)) (* or could write: else swap pr *)
```

In `sort_pair`, we clearly build and return a new pair in the else-branch, but in the then-branch, do we return a *copy* of the pair referred to by `pr` or do we return an *alias*, where a caller like:

```
val x = (4,3)
val y = sort_pair x
```

would now have `x` and `y` be aliases for the *same* pair? The answer is *you cannot tell* — there is no construct in ML that can figure out whether or not `x` and `y` are aliases, and no reason to worry that they might be. *If* we had mutation, life would be different. Suppose we could say, "change the first part of the pair `x` is bound to so that it holds 5 instead of 4." Then we would have to wonder if `#1 y` would be 4 or 5.

In case you are curious, we would expect that the code above would create aliasing: by returning `pr`, the `sort_pair` function would return an alias to its argument. That is more efficient than this version, which would create another pair with exactly the same contents:

```
fun sort_pair (pr : int*int) =
```

```
    if (#1 pr) > (#2 pr)
    then (#1 pr, #2 pr)
    else ((#2 pr),(#1 pr))
```

Making the new pair `(#1 pr, #2 pr)` is bad style, since `pr` is simpler and will do just as well. Yet in languages with mutation, programmers make copies like this all the time, exactly to prevent aliasing where doing an assignment using one variable like `x` causes unexpected changes to using another variable like `y`. In ML, no users of `sort_pair` can ever tell whether we return a new pair or not.

Our second example is our elegant function for list append:

```
fun append (lst1 : int list, lst2 : int list) =
    if null lst1
    then lst2
    else hd(lst1) :: append(tl(lst1), lst2)
```

We can ask a similar question: Does the list returned *share* any elements with the arguments? Again the answer does not matter because no caller can tell. And again the answer happens to be yes: we build a new list that "reuses" all the elements of `lst2`. This saves space, but would be very confusing if someone could later mutate `lst2`. Saving space is a nice advantage of immutable data, but so is simply not having to worry about whether things are aliased or not when writing down elegant algorithms.

In fact, `tl` itself thankfully introduces aliasing (though you cannot tell): it returns (an alias to) the tail of the list, which is always "cheap," rather than making a copy of the tail of the list, which is "expensive" for long lists.

The `append` example is very similar to the `sort_pair` example, but it is even more compelling because it is hard to keep track of potential aliasing if you have many lists of potentially large lengths. If I append `[1,2]` to `[3,4,5]`, I'll get *some* list `[1,2,3,4,5]` but if later someone can *change* the `[3,4,5]` list to be `[3,7,5]` is the appended list still `[1,2,3,4,5]` or is it now `[1,2,3,7,5]`?

In the analogous Java program, this is a crucial question, which is why Java programmers *must obsess* over when references to old objects are used and when new objects are created. There are times when obsessing over aliasing is the right thing to do and times when avoiding mutation is the right thing to do — functional programming will help you get better at the latter.

For a final example, the following Java is the key idea behind an actual security hole in an important (and subsequently fixed) Java library. Suppose we are maintaining permissions for who is allowed to access something like a file on the disk. It is fine to let everyone see *who has permission*, but clearly only those that do have permission can actually use the resource. Consider this wrong code (some parts omitted if not relevant):

```
class ProtectedResource {
   private Resource theResource = ...;
   private String[] allowedUsers = ...;
   public String[] getAllowedUsers() {
      return allowedUsers;
   }
   public String currentUser() { ... }
   public void useTheResource() {
      for(int i=0; i < allowedUsers.length; i++) {
         if(currentUser().equals(allowedUsers[i])) {
             ... // access allowed: use it
```

```
            return;
         }
      }
      throw new IllegalAccessExcpetion();
   }
}
```

Can you find the problem? Here it is: `getAllowedUsers` returns an alias to the `allowedUsers` array, so any user can gain access by doing `getAllowedUsers()[0] = currentUser()`. Oops! This wouldn't be possible if we had some sort of array in Java that did not allow its contents to be updated. Instead, in Java we often have to remember to *make a copy*. The correction below shows an explicit loop to show in detail what must be done, but better style would be to use a library method like `System.arraycopy` or similar methods in the `Arrays` class — these library methods exist because array copying is necessarily common, in part due to mutation.

```
public String[] getAllowedUsers() {
   String[] copy = new String[allowedUsers.length];
   for(int i=0; i < allowedUsers.length; i++)
      copy[i] = allowedUsers[i];
   return copy;
}
```