CSE 341, Spring 2008, Lecture 12 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.

This lecture discusses two topics: parametric polymorphism and what it means for two pieces of code to be "equivalent." For both topics, we will take a more conceptual/theoretical view rather than just showing you, "how to do something in ML." As the course progresses, we will have a few other lectures that focus more on concepts and not just on some specific language constructs. This material is important for understanding that the concepts are more essential than just how they appear in one language.

Parametric polymorphism is a fancy name for the "forall types" (i.e., types that mention *type variables* like `'a`) we have been using in ML. While statically typed functional languages have had such types for over two decades, they are now also part of widely used object-oriented languages such as Java and C#.

As an example, consider the ML type `('a*'b)->('b*'a)`, which is the type we could give to a function that takes a pair and returns a pair that swaps the position of the argument's pieces. What this means in ML is, "for all possible types, call them alpha and beta, we have a function that takes a pair of an alpha and a beta and returns a pair of a beta and an alpha." This type affects the caller and the callee:

- The caller has the flexibility to use the function with any two not-necessarily-different types.
- The callee cannot make any assumption about what the types are. An amazing fact about ML is that if a function of type `('a*'b)->('b*'a)` returns, then it must act like the swapping function `fun f(x,y) = (y,x)`. After all, it cannot "do" anything with the pieces of the pair it is given.

To make the key notion of "for all" more explicit, let's write this type as `forall 'a, 'b. (('a*'b)->('b*'a))`. This is not actually ML syntax (there is no way in ML to make the "for all") explicit, but it helps explain the general theory and where ML is limited.

Now, given a "for all type" of the form `forall 'a. (t)` where `t` is a type, we can explain "for all" by saying that such a type can be *instantiated* by replacing it with the type `t2` made from taking `t` and replacing all the `'a` with any type `t3` (assuming there are not shadowing uses of `forall 'a ...` inside `t`). For example, starting with `forall 'a, 'b. (('a*'b)->('b*'a))` and instantiating it with `string` for `'a` and `int->int` for `'b`, we end up with `(string * (int->int)) -> ((int->int) * string)`, which is simply a less-general type than we started with.

We now have a very general way to describe an infinite number of possible types:

- We have a small collection of *base types* like `int`, `string`, and `real`.
- We can build types out of smaller types by making tuple types, function types, datatypes, etc.
- We can make a type like `forall 'a. (t)` where `t` can be any type.

There is nothing wrong with this definition, but ML is not quite this flexible. According to the definition above, we could write a type like

```
(forall 'a. ('a -> ('a*'a))) -> ((int*int) * (bool*bool))
```

This describes a function that takes a polymorphic pair-making function and returns a pair of pairs. In ML, you cannot use "forall" like this. Instead, forall is always *implicit* (you don't write it) and *all the way to the outside left* (it is never part of a larger type). So when you write

```
('a -> ('a*'a)) -> ((int*int) * (bool*bool))
```

that is really saying

```
forall 'a. (('a -> ('a*'a)) -> ((int*int) * (bool*bool)))
```

which is *not quite* the same thing. This type describes a function where the caller will have to instanitate `'a` with *one* type and then pass in a pair-making function for *that* type, rather than passing in a polymorphic function.

This is admittedly a subtle point. To see an example, here is a function that has the first not-in-ML type and does not have the second type:

```
fun f pairmaker = (pairmaker 7, pairmaker true)
```

This function does not type-check in ML. Type inference, which is the real reason ML has this "all the way to the outside left" restriction will reject it because it looks like `pairmaker` has to take an `int` and a `bool`. That would be possible if there were a way to say `f` must be passed a polymorphic function. Then we could call `f` with arguments like `fn y=>(y,y)` but not with `fn y=>(y+1,y)`. This limitation of ML arises rarely in practice, but more often than never.

Let's now compare parametric polymorphism with subtyping like you have seen in Java and we will study near the end of the course. Consider again our swap function and its type:

```
  fun swap (x,y) = (y,x) (* ('a * 'b) -> ('b * 'a) *)
```

Compare that to a plain-old-Java static method that does the same thing:

```
 class Pair {
  Object x;
  Object y;
  Pair(Object _x, Object _y) { x=_x; y=_y; }
  static Pair swap(Pair pr) {return new Pair(pr.y, pr.x);}
 }
```

Ignoring the fact that Java is more verbose and does not have built-in support for pairs, there is still a very big difference between these two approaches. In the ML version, we know the type of the pair returned is not "just `'b*'a` for any two types" but rather that this `'b` and `'a` are the same as the `'b` and `'a` used to describe the argument to `swap`. So if we call `swap(4,"hi")`, we know the result is a `string*int`. This has two advantages:

- Unlike in Java, callers do not have the inconvenience and error-proneness and cost of casting a field of the result from `Object` back to what "we know" it will be.
- Unlike in Java, what "we know" is actually true. There is no way for a broken swap function not to return a `string*int` given an `int*string` assuming it type-checks.

Subtyping has some complementary advantages we'll study later, but using subtyping where what you want is parametric polymorphism is unpleasant and not good style when you have a better choice. For many years in Java, you did not have a better choice than subtyping code like the `Pair` class above, but then Java added *generics*, which is just a synonym for parametric polymorphism. Using generics, our example would look like this:

```
class Pair<T1,T2> {
    T1 x;
    T2 y;
    Pair(T1 _x, T2 _y) { x=_x; y=_y; }
    static <T1,T2> Pair<T2,T1> swap(Pair<T1,T2> pr) {
       return new Pair<T2,T1>(pr.y,pr.x);
    }
}
```

This has all the parametric-polymorphism advantages of the ML version. It is just a lot more verbose syntactically since there is no type inference, we need explicit return statements, etc.

Parametric polymorphism becomes particularly useful for defined functions over containers (lists, arrays, sets, hashtables, etc.) that have elements of the same type. As we have studied, not all functions over lists are polymorphic, but many are, including the constructors:

```
val [] : 'a list
val :: : ('a * ('a list)) -> 'a list (* infix is syntax *)
val map : (('a -> 'b) * ('a list)) -> 'b list
val fold : ('a * 'b -> 'b) -> ('a list) -> 'b
```

What exactly, then is `list`? It is *not* a type; you cannot say a function has type `list->int`, for example. It is a *type-constructor*, something that makes a type out of another type. So `int list` is a type, `'a list` is a type, `(int->int) list` is a type, etc., and so there exist types like `(int->int) list -> int` and so on. Remember for the examples above there is an implicit "for all" on the outside left. For example, the type of `[]` is "an alpha list for all types alpha."

We can define our own type-constructors in ML. It would be a poor language design to have the only type-constructors be built-in features like lists. In general, if a feature is useful for built-in features, it is useful for user-defined things too. To define a type-constructor in ML, we just use a `datatype` binding. We explicitly give one or more type-constructor arguments that we can then use in the types of the constructors. Here are two examples:

```
datatype 'a non_mt_list = One of 'a
                        | More of 'a * ('a non_mt_list)
datatype ('a,'b) mytree =
   Leaf of 'a
 | Node of 'b * ('a,'b) mytree * ('a,'b) mytree
```

The first one is a type for lists that have at least one argument. The second is for trees where leaves have one type of data and internal nodes have a different type of data. Notice `mytree` is not a type, something like `(int,string) mytree` is a type. Fortunately, type inference can still figure out all the types for us, since `Leaf` and `Node` are just constructors with polymorphic types — `Leaf` has type `'a -> ('a,'b) mytree` and `Node` has type `'b * ('a,'b) mytree * ('a,'b) mytree -> ('a,'b) mytree`. Here are 3 expressions and the inferred types:

```
Node("hi",Leaf 17,Leaf 4)     (* (string,int) mytree *)
Node(14,Leaf "hi",Leaf "mom") (* (int,string) mytree *)
(* Node("hi",Leaf 17,Leaf true) *) (* doesn't type-check *)
```

Now that we can define our own type-constructors, there really was no need for ML to build in support for lists at all. Yes, the syntax of writing `::` infix (between its two arguments or subpatterns) and the syntactic sugar of `[e1,e2,...,en]` are nice, but other than that we could just have defined

```
datatype 'a list = Empty | Cons of 'a * 'a list
```

and had everything we needed.

Unfortunately, this is not quite the end of the story on ML parametric polymorphism. Without one additional restriction, ML's mutable references — which are sometimes useful, but we have used only in our callback example when studying higher-order functions — can make the type system *unsound*. An unsound type system does not actually prevent what it claims to prevent, such as treating an `int` as a function or enforcing module signatures. This is an example of a program that demonstrates the problem:

```
val x = ref []       (* 'a list ref *)
val _ = x := ["hi"]  (* instantiate 'a with string *)
val _ = (hd(!x)) + 7 (* instantiate 'a with int -- bad!! *)
```

The rules we studied for type inference would accept this program even though we should not. To prevent this, ML will reject the first line because of something called "the value restriction". The value restriction requires any expression that is given a polymorphic type to be a variable or a value (including function definitions, constructors, etc.). Because `ref []` is not a value, we can give it type `(int list) ref` or `(string list) ref` but not `('a list) ref`. But type inference cannot figure out what non-polymorphic type to give `x` in our example, so either the programmer must supply an explicit type or the binding is rejected by the type-checker. While it's not at all obvious that this simple restriction makes the whole type system sound, it turns out to be enough.

The value restriction does sometimes get in your way even when you are not using mutation (since the type-checker does not *know* you are not using mutation). For example, this completely harmless code is rejected:

```
val pr_list = List.map (fn x => (x,x))
```

As cool as partial application is, if the result would have a polymorphic type, we cannot bind it to a variable due to the value restriction. We can either give an explicit non-polymorphic type or we can use an extra function wrapper so that the expression we are using is already a value. (Recall functions *are* values.) So any of these three approaches work fine:

```
val pr_list : int list -> (int*int) list = List.map (fn x => (x,x))
val pr_list = fun lst => List.map (fn x => (x,x)) lst
fun pr_list lst = List.map (fn x => (x,x)) lst
```

You do not need to know anything about the value restriction really except how to work around it when it comes up.

The idea that one piece of code is "equivalent" to another piece of code is fundamental to programming and computer science. You are informally thinking about equivalence every time you simplify some code or say, "here's another way to do the same thing." Also notice that our use of restrictive signatures in the previous lecture was largely about equivalence: by using a stricter interface, we make more different implementations equivalent because clients cannot tell the difference.

We want a precise definition of equivalence so that we can decide whether certain forms of code maintenance or different implementations of signatures are actually okay. We do not want the definition to be so strict that we cannot make changes to improve code, but we do not want the definition to be so lenient that replacing one function with an "equivalent" one can lead to our program producing a different answer. There are many different possible definitions that resolve this strict/lenient tension slightly differently; we'll just consider one that is useful and commonly assumed by people who design and implement programming languages.

The intuition behind our definition is as follows:

- A function `f` is equivalent to a function `g` (or similarly for other pieces of code) if they produce the same answer and have the same side-effects no matter where they are called in any program with any arguments.

- Equivalence does *not* require the same running time, the same use of internal data structures, the same helper functions, etc. All these things are considered "unobservable", i.e., implementation details that do not affect equivalence.

As an example, consider two very different ways of sorting a list. Provided they both produce the same final answer for all inputs, they can still be equivalent no matter how they worked internally or whether one was faster. However, if they behave differently for some lists, perhaps for lists that have repeated elements, then they would not be equivalent.

However, the discussion above was simplified by implicitly assuming the functions always return and have no other effect besides producing their answer. To be more precise, we need that the two functions when given the same argument in the same environment:

1. Produce the same result (if they produce a result).

2. Have the same (non)termination behavior; i.e., if one runs forever the other must run forever.

3. Mutate the same memory in the same way.

4. Do the same input/output.

5. Raise the same exceptions.

While this list of requirements makes equivalence stricter, they are all important for knowing that if we have two equivalent functions, we could replace one with the other and no use anywhere in the program will behave differently.

Moreover, if you look at requirements 3 and 4, you will see that these are exactly the things that functional programs like ML discourage you from doing. Yes, in ML you *could* have a function body mutate some global reference or something, but it is generally bad style to do so. Other functional languages are *pure functional languages* meaning there really is no way to do mutation inside (most) functions.

If you "stay functional" by not doing mutation, printing, etc. in function bodies as a matter of policy, then callers can assume lots of equivalences they cannot otherwise. For example, can we replace `f(x)+f(x)` with `f(x)*2`? In Java, that can be a wrong thing to do since calling `f` might update some counter or print something. In ML, that's also possible, but far less likely as a matter of style, so we tend to have more things be equivalent. In a purely functional languages, we are guaranteed the replacement does not change anything. The general point is that mutation really gets in your way when you try to decide if two pieces of code are equivalent — it's a great reason to avoid mutation in the first place.

Now that we have defined equivalence, we can discuss three important and very general situations where ML-style functions are equivalent to each other. However, we will delay this discussion until later in the Scheme portion of the course (due to time constraints). Instead, we let's consider "syntactic sugar," a phrase we have used informally for several lectures.

A language construct is syntactic sugar if every use of it can be rewritten in terms of another more basic language construct. Of course, the rewritten version must be equivalent. Recognizing that some construct is syntact sugar helps simplify a language definition because we can just define the language in terms of the program after the rewriting, often called the desugared code. Recall some examples of syntactic sugar we have seen:

- `e1 andalso e2` can be defined as `if e1 then e2 else false`
- `if e1 then e2 else e3` can be defined as `case e1 of true => e2 | false => e3`
- tuples are really records with field names `1`, `2`, ...

Because of possible side-effects or nontermination, it is essential that we think about what expressions the desugared version evaluates in what order. For example, `e1 andalso e2` is *not* equivalent to `if e2 then e1 else false` because the expression `e2` might not terminate.

It is almost the case that `let val p = e1 in e2 end` can be sugar for `(fn p => e2) e1`. After all, for any expressions `e1` and `e2` and pattern `p`, both pieces of code:

- Evaluate `e1` to a value.
- Match the value against the pattern `p`.
- If it matches, evaluate `e2` to a value in the environment extended by the pattern match.
- Return the result of evaluating `e2`.

Since the two pieces of code "do" the exact same thing, they must be equivalent. In Scheme, this will be the case. In ML, the only difference is the type-checker: The variables in `p` are allowed to have polymorphic types in the let-version, but not in the anonymous-function version.

The reason is our earlier discussion that all forall-types have to be on the outside left. For example, consider `let val x = (fn y => y) in (x 0, x true) end`. This silly code type-checks and returns `(0,true)` because `x` has type `'a->'a`. But `(fn x => (x 0, x true)) (fn y => y)` does not type-check because there is no non-polymorphic type we can give to `x` and function-arguments cannot have polymorphic types.