CSE341, Fall 2011, Lecture 13 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 discusses two topics:

- What it means for two piece of code to be "equivalent" and various ways that two functions can be equivalent.
- A more precise description of parametric polymorphism (type variables) and how even ML has limitations in how much it supports generics. (Near the end of the course we will contrast parametric polymorphism with subtyping.)

Neither of these topics involve showing you new constructs in ML that let you do something new or do something more conveniently. Rather, these are more conceptual/theoretical topics that will hopeful improve the way you look at software written in any language (particularly the notion of equivalence).

Why Talk About Equivalence?

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." This kind of reasoning comes up in several common scenarios:

- Code maintenance: Can you simplify, clean up, or reorganize code without changing how the rest of the program behaves?
- Backward compatibility: Can you add new features without changing how any of the existing features work?
- Optimization: Can you replace code with a faster or more space-efficient implementation?
- Abstraction: Can an external client tell if I make this change to my code?

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.

Defining Equivalence

There are many different possible definitions that resolve this strict/lenient tension slightly differently. We will focus on one that is useful and commonly assumed by people who design and implement programming languages. We will also simplify the discussion by assuming that we have two implementations of a function and we want to know if they are equivalent.

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 (visible-to-clients) memory in the same way.
4. Do the same input/output
5. Raise the same exceptions

These requirements 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.

Another Benefit of Side-Effect-Free Programming

If you look at requirement 3, 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 addition to being able to remove repeated computations (like `f(x)` above) when maintaining side-effect-free programs, we can also reorder expressions much more freely. For example, in Java:

```
int a = f(x);
int b = g(y);
return b - a;
```

might produce a different result from:

```
return g(y) - f(x);
```

since the methods get called in a different order. Again, this is possible in ML too, but if we avoid side-effects, it is much less likely. (We might still have to worry about a different exception getting thrown and other details, however.)

General Forms of Equivalence

Equivalence is subtle, especially when you are trying to decide if two functions are equivalent without knowing all the places they may be called. Yet this is common, such as when you are writing a library that unknown clients may use. We now consider several situations where equivalence is guaranteed in any situation, so these are good rules of thumb and are good reminders of how functions and closures work.

First, recall the various forms of syntactic sugar we have learned. We can always use or not use syntactic sugar in a function body and get an equivalent function. If we couldn't, then the construct we are using is not actually syntactic sugar. For example, these definitions of `f` are equivalent regardless of what `g` is bound to:

```
fun f x =
    if x
    then g x
    else false
```

```
fun f x =
    x andalso g x
```

Notice though, that we could not necessarily replace `x andalso g x` with `if g x then x else false`.

Second, we can change the name of a local variable (or function parameter) provided we change all uses of it consistently. For example, these two definitions of `f` are equivalent:

```
val y = 14
fun f x = x+y+x
```

```
val y = 14
fun f z = z+y+z
```

But there is one rule: in choosing a new variable name, you cannot choose a variable that the function body is already using to refer to something else. For example, if we try to replace `x` with `y`, we get `fun y = y+y+y`, which is *not* the same as the function we started with. A previously-unused variable is never a problem.

Third, we can use or not use a helper function. For example, these two definitions of `g` are equivalent:

```
val y = 14
fun g z = (z+y+z)+z
```

```
val y = 14
fun f x = x+y+x
fun g z = (f z)+z
```

Again, we must take not to change the meaning of a variable due to `f` and `g` having potentially different environments. For example, here the definitions of `g` are *not* equivalent:

```
val y = 14
val y = 7
fun g z = (z+y+z)+z
```

```
val y = 14
fun f x = x+y+x
val y = 7
fun g z = (f z)+z
```

Fourth, as we have explained before with anonymous functions, unnecessary function wrapping is poor style because there is a simpler equivalent way. For example, `fun g y = f y` and `val g = f` are always equivalent. Yet once again, there is a subtle complication. While this works when we have a variable like `f` bound to the function we are calling, in the more general case we might have an *expression* that evaluates to a function that we then call. Are `fun g y = e y` and `val g = e` always the same for any *expression* `e`? No.

As a silly example, consider `fun h() (print "hi" ; fn x => x+x)` and `e` is `h()`. Then `fun g y = (h()) y` is a function that prints every time it is called. But `val g = h()` is a function that does not print — the program will print `"hi"` once when creating the binding for `g`. This should not be mysterious: we know that

val-bindings evaluate their right-hand sides "immediately" but function bodies are not evaluated until they are called.

A less silly example might be if `h` might raise an exception rather than returning a function.

Fifth, 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 Racket, this will be the case (with different syntax). 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.

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. This is just how type-inference works in ML and is related to the discussion of parametric polymorphism below.

Revisiting our Definition of Equivalence

By design, our definition of equivalence ignores how much time or space a function takes to evaluate. So two functions that always returned the same answer could be equivalent even if one took a nanosecond and another took a million years. In some sense, this is a *good thing* since the definition would allow us to replace the million-year version with the nanosecond version.

But clearly other definitions matter too. Course in data structures and algorithms study *asymptotic complexity* precisely so that they can distinguish some algorithms as "better" (which clearly implies some "difference") even though the better algorithms are producing the same answers. Moreover, asymptotic complexity, by design, ignores "constant-factor overheads" that might matter in some programs so once again this stricter definition of equivalence may be too lenient: we might actually want to know that two implementations take "about the same amount of time."

None of these definitions are superior. All of them are valuable perspectives computer scientists use all the time. Observable behavior (our definition), asymptotic complexity, and actual performance are all intellectual tools that are used almost every day by someone working on software.

Parametric Polymorphism

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 (`fun swap (x,y) = (y,x)`). 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 any function of type `('a*'b)->('b*'a)` returns, then the value returned is the same value that `swap` returns.

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`. 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.

In theory, we could use "for all types" as smaller parts of larger types as in

```
(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 instantiate `'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.

So why does ML have this restriction? Mostly because there are always trade-offs in language design and with this restriction type inference is much more straightforward. But more generally, every sound type system has "unnecessary" restrictions, meaning it rejects programs that "do nothing wrong". This is because for very basic definitions of "wrong" like "never try to treat a string like it is a function," rejecting *exactly* the programs that might do this is *impossible* for an algorithm. (And algorithm here is something that must terminate and produce the correct answer for every program.) This is the notion of *undecidability* that you study in another course, usually starting with the halting problem.