CSE 341, Spring 2011, Lecture 24 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 covers several topics related to type systems for OO languages and how they relate to other topics:

- Comparing named types to structural types
- Contrasting types and classes, and why languages like Java, C#, and C++ choose to combine the concepts
- Comparing parametric polymorphism to subtyping
- Combining parametric polymorphism to subtyping, and the occasional usefulness of bounded polymorphism

Named types vs. structural types

The previous lecture considered types for objects where the types simply listed the types of fields and/or methods that objects with that type would have. We used these types to consider subtyping. However, in languages like Java, types don't look like this. Instead, they look like the names of classes or interfaces. So we write a type like `C` and then look at the definition of some class `C` to see what fields/methods objects with type `C` would have. We say languages like Java have *named types* (or *nominal typing*) as opposed to *structural types* (or *structural typing*). This makes sense: Does a type simply name something or does it describe an object's structure?

Everything we learned about subtyping with structural types also applies to named types. If we want `C` to be a subtype of `D`, then we must be able to treat every object of type `C` as though it had type `D`, with covariant method results, contravariant method arguments, etc.

With named types like in Java, we only have the subtyping that is explicitly declared by the programmer (plus transitivity). For example, given

```
class C extends D implements I,J { ... }
```

the only supertypes of the type `C` are `D`, `I`, `J`, and the supertypes of `D`, `I`, and `J`. This allows fewer supertypes than structural typing. After all, suppose `C` describes objects that have an `x` field of type `int` and a `y` field of type `int`. It would be sound to let `C` be a subtype of some "unrelated" type `Foo` that happens to promise nothing other than an `x` field of type `int`.

The argument between structural typing and nominal typing has been waged for decades and is unlikely to be resolved. Proponents of structural subtyping point out that it allows more code reuse (in our example above, passing a `C` to a method expecting a `Foo` for example) while remaining sound. Moreover, with structural subtyping, when one creates a new interface type, one does not have to go back to class definitions that already exist to add the interface to the class definition.

Proponents of nominal typing claim that less code reuse catches more bugs. For example, if cowboys and artists both have `draw` methods, some method that only uses an argument's `draw` method and intends to paint a picture would in fact take both cowboys and artists under structural subtyping. Having fewer subtypes can also make field and method lookup a little faster in the code generated by a compiler. (We won't describe why since this isn't a compilers course.) Finally, since a class-based OO language requires you to write down class definitions anyway, using named types based on these class definitions saves programmers the trouble of writing down structural types.

However, classes and types are not the same thing...

Classes vs. Types

Many languages confuse classes and types. For example, in Java every class is a type, and most types are classes. (Other types include `int`, array types, and interfaces.) Moreover, we require that a subclass is a subtype: If `C` extends `D`, then `C` is a subtype of `D`. Conversely, the only subtyping that is not the result of subclassing is from implementing interfaces.[1]

So are classes and types the same thing? No!

A class is a run-time concept. Every object has a class. An object's class determines how it responds to messages. Subclassing inherits behavior from the superclass, saving code duplication and enabling specialization via dynamic dispatch.

A type is a static ("compile-time") concept. Every expression has a type. An expression's type describes what operations can be performed without the program causing an error the type system is supposed to prevent. Subtyping is about substitutability, as studied in the previous lecture.

So why do languages like Java confuse subclassing and subtyping? Because it's almost always what you want. But not quite always. It might be nice to have a subclass that was not a subtype. For example, consider a 2DPoint class with methods:

```
double get_x() { ... }
double get_y() { ... }
void set_x(double v) { ... }
void set_y(double v) { ... }
double distBetween(2DPoint other) { ... }
...
```

A subclass `3DPoint` could inherit the first four methods and override `distBetween`, having it take a `3DPoint` instead:

```
double distBetween(3DPoint other) { ... }
```

However, if a subclass must be a subtype such overriding is *wrong* because method arguments are contravariant. If we took a `3DPoint`, subsumed it to a `2DPoint` and called its `distBetween` with another `2DPoint`, the type system would allow it but the body of `3DPoint`'s `distBetween` would be called with an argument of the wrong class. We would like instead in this example just to disallow subsuming a `3DPoint` to a `2DPoint`. This is not possible in Java, but that doesn't mean it isn't useful.

(In actual Java, declaring `distBetween` to take a `3DPoint` in the subclass does not actually cause overriding. Instead, instances of `3DPoint` then have two `distBetween` methods, one used for instances of `2DPoint` and one used for instance of `3DPoint`. However, which method gets called is decided statically, based on the compile-time type of the argument at any call-site. We won't have time to study this idea, called *static overloading*, carefully. So since calling the `2DPoint distBetween` on a `3DPoint` may make very little sense, we may want to override `double distBetween(2DPoint other)` to raise an exception. However, this still won't catch the error at compile time.)

Conversely, we may want a subtype that is not a subclass. For example, different classes may have `display` methods that take some window and put a picture in it. They may not share any code; we simply want a type that describes objects with such methods. In Java, one would use an interface like:

```
interface Displayable {
  void display(Window w);
}
```

Implementing interfaces are exactly the way Java programmers create subtypes without using subclassing.

Having "untangled" the difference between classes and types, we can also better understand Java's abstract methods and classes. Because abstract methods provide no behavior, they really have nothing to

[1] Well, there is also covariant array subtyping, but as discussed in section, this is a controversial decision that contradicts what we studied with subtyping.

do with subclassing and inheritance. They are entirely a typing thing. An abstract method ensures: (1) All non-abstract subclasses override the method, and (2) No instances of a class with abstract methods are created. These are compile-time checks (i.e., part of the type system). The reason to prefer abstract methods over non-abstract methods that simply raise exceptions is to get this compile-time check. This reason is significant, but there is no other reason.

Parametric polymorphism vs. subtyping

Scheme and Ruby really are not that different beyond Scheme having more parentheses and Ruby having dynamic dispatch by default (rather than something the programmer has to code up). ML and Java are much more different because their type systems take complementary approaches to supporting code reuse while still being sound. ML uses parametric polymorphism (see lecture 12, in short those `'a` and `'b` types) whereas Java uses subtyping. These two type system features are complementary. Neither is a good substitute for the other. That is exactly why Java eventually added *generics* (a synonym for parametric polymorphism).

Consider various ML functions with polymorphic types:

```
compose: ('a -> 'b) * ('b -> 'c) -> ('a -> 'c)
isempty: 'a list -> bool
map: ('a -> 'b) * 'a list -> 'b list
```

These types are sound, expressive (they let callers use the functions for arguments of various types), and convenient. For convenience, notice that when we call `map` with arguments of type `int->bool` and `int list`, we do not need any casts on the arguments or the result, which the type-checker determines is `bool list`. Subtyping is not a good substitute here. Our only reasonable choice for the type of `map` in a language like Java (ignoring generics) is something like

```
(Object -> Object) * Object list -> Object list
```

But now using the function is inconvenient and error-prone. Given an `int->bool`, we would need to wrap it with an `Object->Object` that performed various casts. Similarly, the result is not the type we want, so we would have to extract elements from the list and then cast them to `bool`. These casts actually have run-time cost. Also, if we get them wrong we may get a run-time error, i.e., the type system is not really helping us. Also, perhaps the implementation of `map` is wrong and does not actually return a list with elements of the types we expect. None of these shortcomings arise with parametric polymorphism.

In other examples, subtyping is exactly what you want. For example, consider a function that determines if a `2DPoint` is in the right half of the plane:

```
boolean isXPos(2DPoint p) { return p.x > 0; }
```

Subtyping makes it very convenient to call `isXPos` with a subtype of `2DPoint` (such as `3DPoint`). One can encode similar ideas with type variables, but subtyping still captures the idea quite directly.

Subtyping is also convenient for using objects like closures since subtypes can have "more" "private fields" than the supertype promises they have. Here is a simple example where the interface `J` just describes anything with a function `f` that takes and returns an `int` but subclasses can have other fields and methods:

```
interface J { int f(int); }
class MaxEver implements J {
   private int m = 0;
   public int f(int i) { if(i > m) m = i; return m; }
}
```

If you have a language with parametric polymorphism and subtyping (such as Java as of version 1.5), you can program with the benefits of both. In fact, you sometimes want to combine the two ideas with *bounded polymorphism*. Whereas a type like `'a` means "for all types `'a`" and a type like `C` means "the type `C`, which could actually be any subtype of `C`", a bounded type says, "for all types `'a` that are subtypes of type `Foo`."

This is more powerful than just using `Foo` or just using `'a`. As a simple example consider a function of type, "for all `'a` that are subtypes of `Foo`, take an `'a` and return an `'a`." The body can assume the argument has everything a `Foo` has. And if some caller passes a `Bar` that is a subtype of `Foo`, then the caller knows the result is a `Bar`, not just a `Foo`. A call-site that passed an argument that was not a subtype of `Foo` would be rejected as a type error.

The code associated with this lecture considers a simple linked-list library example in Java. The first version uses subtyping, which is not the right thing; it ends up using casts that can fail at run-time. The second version uses parametric polymorphism, which works better but the `copy` method still has a less permissive type than we would like. The third version uses bounded polymorphism to give `copy` a more useful type. Note that `List<B>` cannot be a subtype of `List<A>` even if `B` is a subtype of `A` because our lists our mutable. This relates to the field subtyping in the previous lecture.