CSE341, Fall 2011, Lecture 26 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 applies the general concepts of record and function subtyping from the last lecture to object-oriented languages, mostly Java. We will see that:

- Record and function subtyping are excellent guides to what should be allowed in statically typed object-oriented languages to prevent "field missing" and "method missing" errors.
- But Java has two key differences worth exploring:
  - It uses class and interface *names* for types, which is convenient and/but prevents some subtyping that would be sound.
  - It does not support contravariant arguments on methods. Instead it supports *static overloading* based on the types of method arguments.
- `self`/`this` is special with respect to subtyping: It is covariant in subclasses even though we sometimes think of it as an extra argument to methods.

Basic Subtyping for Objects and Java's Restrictions Thereof

An object is basically a record holding fields (which we assume here are mutable) and methods. We assume the "slots" for methods are immutable: If an object's method `m` is implemented with some code, then there is no way to mutate `m` to refer to different code. (An instance of a subclass could have different code for `m`, but that's different than mutating a record field.)

With this perspective, sound subtyping for objects follows from sound subtyping for records and functions:

- A subtype can have extra fields.
- Because fields are mutable, a subtype cannot have a different type for a field.
- A subtype can have extra methods.
- Because methods are immutable, a subtype can have a subtype for a method, which means the method in the subtype can have contravariant argument types and a covariant result result type.

That said, object types in Java and C# do not look like record types and function types. For example, we cannot write down a type that looks something like:

```
{fields : x:real, y:real, ...
 methods: distToOrigin : () -> real, ...}
```

Instead, we reuse class names as types where if there is a class `Foo`, then the type `Foo` includes in it all fields and methods implied by the class definition (including superclasses). And, as discussed previously, we also have interfaces, which are more like record types except they do not include fields and we use the name of the interface as a type. Subtyping in Java and C# includes only the subtyping explicitly stated via the subclass relationship and the interfaces that classes explicitly indicate they implement (including interfaces implemented by superclasses).

All said, this approach is more restrictive than subtyping requires, but since it does not allow anything it should not, it soundly prevents "field missing" and "method missing" errors. In particular:

- A subclass can add fields but not remove them
- A subclass can add methods but not remove them
- A subclass can override a method with a covariant return type
- A class can implement more methods than an interface requires or implement a required method with a covariant return type

Shadowing Fields

Because fields are mutable, we know a subtype cannot have a different type for a field than a supertype. So it is surprising that Java actually allows a class to declare a field with the same name as a field already declared in a superclass. For example, this code is allowed in Java:

```
class Color { String colorName; }
class FancyColor extends Color { double shade; }
class Point { ... }
class ColorPoint extends Point {
  Color color;
  ...
}
class MyColorPoint extends ColorPoint {
  FancyColor color;
  ...
}
```

It would not be sound to allow the `MyColorPoint` class to change the `color` field to hold a `FancyColor` and have code assume this. After all, methods inherited from `ColorPoint` could assign `color` to hold instances of `Color` leading to errors if we then tried to read the `shade` field.

The conundrum is actually easily solved: In Java, the declaration `FancyColor color` adds a *different field* to instances of `MyColorPoint` that happens to have the same name as another field. Since it is a new, different field, it can have any type whatsoever, not necessarily related to the type of the other field named `color`.

Field look-ups do *not* use dynamic dispatch, so any uses of `color` in methods of `ColorPoint` will still use the `color` field of type `Color`. In `MyColorPoint`, the new field shadows the old one, so `color` will refer to the new field. To access the old field, code can use `super.color`.

Static Overloading

A similar but more common and convenient weirdness relates to Java's support for having multiple methods with the same name. Even though it would be sound, there is simply no way in Java to override a method with contravariant arguments. Instead, Java works like this:

If a class declares a method with name `m`, return type `t0` and argument types `t1`, `t2`, ..., `tn` (in that order), then:

- If the superclass already has a method name `m` with argument types `t1`, `t2`, ..., `tn` (in that order), then this is overriding. The type-checker requires `t0` to be the same as the return type in the superclass method or a subtype (covariant subtyping on result types).
- Else this is a new method added to the subclass, not overriding, even if the superclass (or the subclass) has other methods named `m`.

Having multiple methods with the same name can be convenient so that you do not have to think up different names for similar behavior. For example, a `Rational` class could have methods like:

```
void add(Rational r) {...}
void add(int i) {...}
void add(double d) {...}
```

But you must take care in Java to make sure you are overriding or not overriding when you intend to. It can be confusing to change or reorder argument types accidentally and, as a result, have different methods with the same name.

More importantly, we need to define which method gets called when there are multiple methods with the right name. That is, we must revise our definition of the most fundamental issue in object-oriented programming: What is the semantics of `e0.m(e1,...,en)`? As before, we start as follows:

- Evaluate `e0`, ..., `en` to objects `v0`, ..., `vn`.
- Use the (run-time) class of `v0` to look up `m` (dynamic dispatch).

But now there may be multiple choices `m`. Java picks the "best" choice using the *static types* of `e1`, ..., `en`, **not** the classes `v0`, ..., `vn`. This semantics is called *static overloading* since we overload the name `m` to mean multiple things and disambiguate using the static types of the arguments. It is *not* dynamic dispatch on the arguments, which would be a different semantics called *multimethods*. There are programming languages with multimethods, but Java and C# do not have them.

The exact definition of "best"choice is remarkably complicated and we will not explain all the rules, but we will give the rough idea and some examples. Simple cases like the `add` methods for `Rational` are easy: Type-checking the argument should produce an `int`, `double`, or `Rational` and the "best" choice should be obvious. But even here we allow a subtype of `Rational` (using subtyping to pick the method that takes a `Rational`) or other numeric types (using a conversion to pick either the `int` version or the `double` version according to rules spelled out in the language definition). Also note that we always pick a method with the correct number of arguments. And if there is no choice that would type-check, the call, as usual, does not type-check.

Here is a more complicated example where we have 5 methods named `m` and the comments next to the methods and example method calls indicate which method will get used by each call:

```
class Color extends Object { String s; }
class FancyColor extends Color { double shade; }

class MyClass {
  void m(Object x)     {...} // A
  void m(Color x)      {...} // B
  void m(FancyColor x) {...} // C
  void m(Color x, FancyColor y) {...} // D
  void m(FancyColor x, Color y) {...} // E
}
MyClass   obj = new MyClass(...);
Color      c1 = new Color(...);
FancyColor c2 = new FancyColor(...);
Color      c3 = new FancyColor(...); // subtyping!
obj.m(c1);    // B
obj.m(c2);    // C
```

```
obj.m(c3);    // B static overloading!
obj.m(c1,c2); // D
obj.m(c1,c3); // type error: no method matches
obj.m(c2,c2); // type error: no best match (tie)
```

We now explain each of the six calls in more detail:

- `obj.m(c1)` calls `m` with one argument of type `Color`. Either method `A` or `B` would type-check, but since `B` requires "less" subtyping, it is picked.
- `obj.m(c2)` is similar: it calls `m` with one argument of type `FancyColor`. Even though `A`, `B`, and `C` all type-check, `C` requires the "least" subtyping.
- `obj.m(c3)` is handled exactly like `obj.m(c1)` because `c1` and `c3` both have type `Color`. In Java, it is irrelevant that the result of evaluating `c3` is an instance of the `FancyColor` class. Under static overloading, we still pick method `B`.
- `obj.m(c1,c3)` calls `m` with two arguments of type `Color`. This does not type-check because none of the methods can take such arguments: the two-argument methods both require at least one argument to be a `FancyColor`. Again the fact that evaluating `c3` would produce an instance of `FancyColor` class is irrelevant.
- `obj.m(c2,c2)` does not type-check for a fundamentally different reason: Methods `D` and `E` would both type-check and since they "tie" in the amount of subtyping, the type-checker rejects the call rather than break the tie. We can break the tie ourselves by rewriting the call as either `obj.m((Color)c2,c2)`, which would call `D`, or as `obj.m(c2,(Color)c2)`, which would call `E`. Or we could add another method `m` to `MyClass` that takes two `FancyColor` arguments.

Names vs. Structure

Remember that the type systems in Java and C# use class and interface names as types and allow subtyping only as indicated by explicit subclassing and implementation of interfaces. So, for example, the type `ThreeActPlay` is not a subtype of `StringPair` even though it would be fine from the perspective of record subtyping:

```
class StringPair {
 String first;
 String second;
 void setFirst(String x) { ... }
}
class ThreeActPlay {
 String first;
 String second;
 String third;
 void setFirst(String x) { ... }
}
```

To have `ThreeActPlay <: StringPair` we would have to make `ThreeActPlay` an explicit subclass. Whether we *want* such subtyping is disputable. On the one hand, it would let us reuse some code for `StringPair` objects without any risk of a "method missing" error. On the other hand, `ThreeActPlay` and `StringPair` are rather separate concepts and it can be a virtue to have the type-checker given an error if we (accidentally?) use an expression of one type where we expect the other.

Classes vs. Types

Classes and types are different things! Java and C# purposely confuse them as a matter of convenience, but you should keep the concepts separate. A class defines an object's behavior. Subclassing inherits behavior, modifying behavior via extension and override. A type describes what fields an object has and what messages it can respond to. Subtyping is a question of substitutability and what we want to flag as a type error. It is often grating to the ears of a programming-languages expert to hear things like, "overriding the method in the supertype" or, "using subtyping to pass an argument of the superclass." That said, this confusion is understandable in languages where every class declaration introduces a class and a type with the same name.

If, like in Ruby, subclasses did not have to be subtypes, then we could allow overriding to change the types of methods. But this is a dangerous practice: If the overridden method is used by inherited methods of the superclass, then those methods might no longer type-check in the subclass.

Conversely, subtypes do not have to subclasses given features like interfaces, but we still require class declarations to indicate what interfaces are supertypes. There is no fundamental reason for this. For example, if the `Launchable` interface had just one method requirement `void launch();`, then we could implicitly let any class with such a method have its type be a subtype of `Launchable`. This could allow more code, but could lead to errors, e.g., if we do not want to allow a `Missile` to be passed to a method that takes a `Launchable`.

As discussed in a previous lecture, abstract methods, like interfaces, are related to the "type part" of a class declaration not the "behavior part." An abstract method indicates that all values of the type will have such a method, but it provides no behavior for subclasses to inherit.

Covariant `self/this`

As a final subtle detail and advanced point, Java's `this` is treated specially from a type-checking perspective. When type-checking a class `C`, we know `this` will have type `C` or a subtype, so it is sound to assume it has type `C`. In a subtype, e.g., in a method overriding a method in `C`, we can assume `this` has the subtype. None of this causes any problems, and it is essential for OOP. For example, in class `B` below, the method `m` can type-check only if `this` has type `B`, not just `A`.

```
class A {
  int m(){ return 0; }
}
class B extends A {
  int x;
  int m(){ return x; }
}
```

But if you recall our manual encoding of objects in an earlier lecture, the encoding passed `this` as an extra explicit *argument* to a method. That would suggest *contravariant* subtyping, meaning `this` in a subclass could not have a *subtype*, which it needs to have in the example above.

It turns out `this` is special in the sense that while it is like an extra argument, it is an argument that is covariant. How can this be? Because it is not a "normal" argument where callers can choose "anything" of the correct type. Methods are always called with a `this` argument that is a subtype of the type the method expects.

This is the main reason why coding up dynamic dispatch manually works much less well in statically typed languages, even if they have subtyping: You need special support in your type system for `this`.