CSE 341, Spring 2008, 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 considers developing a static type system for an object-oriented language. We won't consider any specific type system for any specific language. Rather, we will consider the general principles and "theory" that would go into developing such a type system. Next lecture we will "connect" this discussion to classes (as in Java) and how it compares to an ML-style type system. The main points of this lecture are:

- An object's type should indicate what methods can be called on it so that there are no message-not-understood errors.
- To avoid being overly restrictive, subtyping — allowing all expressions of one type to also have another type — is extremely useful.
- Width and permutation subtyping are sound.
- Depth subtyping is unsound if fields are mutable.
- Subtyping on methods (or functions) allows contravariant arguments and covariant results. (These terms are defined below.)

Recall the purpose of a type system is to prevent certain errors for all programs, rejecting some programs before they run. In ML, the errors we prevented included treating a string as a function or accessing functions made private by a signature. For pure object-oriented languages, "all we do" is create objects and send them messages (i.e., call methods), so what might we want to prevent?

1. We want to prevent sending a message to an object that cannot understand the message (i.e., has no method of the right name). This is a message-not-understood error.
2. If our language allows methods with multiple arguments, we want to prevent calling a method with the wrong number of arguments.

Because there isn't much else our program does, there isn't much else to prevent. We should also prevent accessing non-existent fields, but if we imagine field accesses as using getter/setter methods, then the first item above covers this case.

(Though we won't have time to discuss it, *static overloading* and/or *multimethods* raise additional errors we may wish to prevent. With these features, languages can have multiple methods with the same name that are actually different methods and we have to decide at each call-site which method is the "best match" for the arguments. If there are situations where there is "no best match", that is an error. Java is an example of a language with static overloading where the type system rejects programs that have calls with no-best-match.)

Recall a type system is *sound* if, by definition, it never accepts a program that could have one of the errors the type system is supposed to prevent. Our goal is to identify the key features a sound type system for objects needs. However, preventing message-not-understood often turns out to be a bit too strong in practice, so many languages make the choice not to catch errors resulting from some "special thing" like Java's `null` or Ruby's `nil`. These are not quite the same: Ruby's `nil` is an object that just happens to respond to very few messages. `null` is not an object, it responds to no messages and evaluating `e.m(e1,...,en)` where `e` evaluates to `null` throws (as you have no doubt seen) a `NullPointerException`. In either case, though, the type of `nil` or `null` should be something that accepts very few messages, but for convenience the type system treats as something that can have *any type*, meaning it can accept *any message*. Since this is not actually true, for our type system to be sound we have to change our definition of what it intends to prevent: We want to prevent a message-not-understood error *unless* the receiver is `null`/`nil` (in which case a run-time error is allowed).

Before considering objects with arbitrary methods, let's just consider objects that are like ML records. They have fields and for every field we assume there is a getter method and a setter method. We will also

assume our language has integers, though we could get by without it. So given an expression like `e.x` or like `e.x = 17`, we need that `e` type-checks and has a field named `x`. That way the expressions above will not have message-not-understood (or perhaps "field-not-understood") errors.

However, it is not enough to let an object's type just list the fields of the object. Consider `obj.x.y`. If all we know about `obj` is that it has fields `x`, `f`, and `a`, then we know `obj.x` is okay, but we do not know if the resulting object has a `y` field. Therefore, like ML record types, we need object types to list fields and the types of the fields. So the definition of types is recursive, as we might expect. Since we are not talking about any specific programming language, we need to make up some syntax for writing down types. We won't use class names since that is largely a separate issues. Instead we'll just write down types "directly". For example, an object with two fields `x` and `y` that both hold numbers could have type:

```
{ x : int, y : int }
```

We can also let ourselves give names to types, which is convenient and necessary for describing lists or trees. Here are some examples:

```
intList = { hd : int, tl : intList }
string = { hd : char, tl : string }
name = { first : string, last : string }
```

In the last example, we describe objects with `first` and `last` fields, both of which hold objects that themselves have `hd` and `tl` fields. We can also write types just like this:

```
{ a : {}, b : { c : int } }
```

This type describes objects with fields `a` and `b` where the contents of the `a` field is an object with no fields and the contents of the `b` field is an object with a `c` field whose contents is a number.

These types are sufficient for defining a sound type system as you might expect. For example, we could allow a variable declaration such as:

```
{ a : {}, b : { c : int } } o1 = [ a => [], b => [ c => 17 ]]
```

where here we are making up syntax for creating objects. We are just writing the name of a field and then an `=>` and then the field's contents, which can be another object. The syntax does not really matter. We could then allow setting a field with another object of the appropriate type:

```
{ c : int } o3 = [ c => 42 ]
o1.b = o3
```

However, requiring the right-hand side of an assignment to have a type exactly equal to the type of the assigned-to location is unnecessarily restrictive. Consider:

```
{ c : int, d : int } o4 = [ c => 42, d => 43 ]
o1.b = o4
```

If we require equal types, we reject this program because `o1.b` has type `{ c : int }` and `o4` has type `{ c : int, d : int }`. However, this assignment is okay: it cannot lead to a message-not-understood error. The object we put in `o1.b` has everything the type of `o1.b` requires, namely a `c` field holding an `int`. If we could somehow enrich our type system so that expressions of type `{ c : int, d : int }` could *also* have type `{ c : int }`, then `o4` could have type `{ c : int }` and the assignment would type-check.

Subtyping is exactly the type-system feature that lets us say, "anything that has some type `t1` can also have type `t2`." We call `t1` the *subtype*. We call `t2` the *supertype*. And we call the rule that lets us give an expression of type `t` some other type that is a supertype of `t` *subsumption*.

We have to choose our rules for what is allowed to be a subtype of what carefully so that subsumption does not break the soundness of our type system. Therefore, *subtyping is not a matter of opinion*. We can only let `t1<:t2` (a short way of writing `t1` is a subtype of `t2`) if any value of type `t1` could be viewed as a value of type `t2`.

Our example above used "width subtyping," which recognizes that an object with more fields is a subtype of an object with fewer fields. (This may sound backwards, but the way to make it sound forwards is to say that there are fewer objects with a `c` field and a `d` field than there are objects with a `c` field. In fact, there is a subset relationship.) In general, for any field names `x1`, ..., `xn`, `y`, and types `t1`, ..., `tn`, `t` we have:

```
{x1:t1 ... xn:tn y:t} <: {x1:t1 ... xn:tn}
```

This rule would only let us "forget" one field, but subtyping is also always *transitive*: If `t1<:t2` and `t2<:t3` then `t1<:t3`. Subtyping is also reflexive: every type is a subtype of itself.

Finally, the rule above suggests we can only forget the "last field," but we can also have *permutation subtyping*, which just says that the order we write down fields in a type need not matter. So if we have some method taking an argument of type `{ c : int, d : int}` and we have an expression of type `{ d : int, c : int}`, permutation subtyping would let us pass the expression in the method call.

Now consider subtyping *inside* another object type. This is called *depth subtyping*. It is tempting to allow this rule:

```
If ti <: t
then {x1:t1 ... xi:ti ... xn:tn} <: {x1:t1 ... xi:t  ... xn:tn}
```

This would be useful if, for example, some method expected an argument of type `{a : { c : int }` and we had an argument of type `{a : { c : int, d : int} }`. Unfortunately, this rule is not sound! It would be sound *if* we knew the body of the method would only get the `a` field of its argument and not set it. Here is an example showing the problem:

```
t1 = {x:{} y:{z:{}}}
t2 = {x:{} y:{}}
line 1: t1 o1 = [x=>[], y=>[z=>[]]]
line 2: t2 o2 = o1  # use _broken_ notion of subsumption
line 3: o2.y = []
line 4: o1.y.z # message not understood!
```

Line 1 initializes `o1` with an object of the correct type for `t1`. Line 2 uses depth subtyping to assign `o1` to an object of type `t2`. Notice `t2<:t1` only if we allow our depth subtyping rule. Also notice `o1` and `o2` are now aliases; they refer to the same object. Line 3 type-checks because `[]` and `o2.y` both have type `{}`. Line 4 type-checks because `o1.y` has type `{z:{}}`. However, running the program produces a message-not-understood error because line 3 mutated the `y` field of the object `o1` (and `o2`) point to to hold `[]`, which has no `z` field.

There are two simple solutions that restore soundness:

1. Get rid of depth subtyping.
2. Get rid of mutation on fields.

So again, banning mutation has a benefit; it allows more subtyping. However, most OO languages do allow fields to be updated, so the more common solution is the first one. That means that if a subtype and a supertype both have the same field, then that field must have the same type in both.

Now let's consider subtyping in the presence of objects with methods. Arbitrary methods are more general than just fields, which we can think of as two methods (a getter and a setter). Methods take arguments and return results. We'll assume an object has any number of methods that are immutable. So the question is when can we allow:

```
{... t0 m(t1,...,tn) ...} <: {... t0' m(t1',...,tn') ...}
```

In other words, if the subtype and supertype both have some method `m` how must the argument and return types for `m` compare to each other between the subtype and the supertype. One sound answer would be to require them to be the same (so `t0'=t0`, `t1'=t1`, ... `tn'=tn`), but that is unnecessarily restrictive.

First consider the return types `t0` and `t0'`. We can allow `t0<:t0'`. For example, suppose `t0={c:int, d:int}` and `t0'={d:int}`. Then if we have an object of type `{t0 m()}` (i.e., an object with one method `m` returning

a t0), we subsume this to {t0' m()} and then we call m, the type-checker will ensure we assume the result has type {d:int}. It actually has type {c:int, d:int}, but it is sound to *assume less* about the result.

Now consider the less intuitive issue of argument types. Can we allow subtyping like we did for argument types? NO!!! Consider this example:

```
{ int m({}) } o1 = ...
{ int m({x : int}) } o2 = ...
o1 = o2
o1.m([]) # calls a method that might do use argument's x field in its body!
```

If o2 has a method expecting an argument of type {x:int}, it is unsound to call that method with [], but that would be allowed if we could subsume { int m({x:int}) } to { int m({}) }. In general, for {... t0 m(t1,...,tn) ...} <: {... t0' m(t1',...,tn') ...} we *cannot* allow some argument (e.g., t1) be a subtype of the supertype's argument (e.g., t1'). However, it *is* sound to allow some argument (e.g., t1) to be a *supertype* of the supertype's argument (e.g., t1'). Consider this similar but crucially opposite example:

```
{ int m({}) } o1 = ...
{ int m({x : int}) } o2 = ...
o2 = o1
o2.m([x=7]) #fine! actual argument has x field that won't be used
```

The type of o2 says m must be called with an argument that has an x field. In fact, o2 refers to an object of type {int m({})} so m is in fact not going to acces the x field. Still, there is no harm in passing an object that does have an x field. In general, a supertype can *require more of method arguments* than the subtype.

Notice the beautiful if unintuitive symmetry: The type of a method in the supertype requires more of its arguments and promises less about its result. That makes sense: the subtype, which is closer to what an object of with the subtype "actually has" can always be conservatively treated as the supertype. After subsumption, only fewer calls to the method will work (it will be harder for arguments to type-check and there is less we can do with the result), so the subsumption cannot lead to a message-not-understood error.

While we have considered method subtyping for object-oriented languages, a functional language with subtyping would work the same way. (ML does not have subtyping, but that's because it complicates type inference.) When can we allow t1->t2 <: t3->t4? The answer is when t2 <: t4 and t3 <: t1. Again, this means the supertype requires more of its argument and assumes less about its result.

There is some jargon for this. We say argument types are *contravariant*, which means the subtype relationship for the arguments is the opposite of what it is for the overall type. We say result types are *covariant*, which means the subtype relationship for the results is the same as what it is for the overall type.

Argument contravariance is the least intuitive concept in the course, but it is worth burning into your memory so that you do not forget it. Many people get confused because it is *not* about calls to methods/functions. Rather it is about the methods/functions themselves. If I have a function of type t1->t2 I can *call* it with a subtype of t1. For example if t3<:t1 and e1 has type t1->t2 and e2 has type t3, then e1(e2) type-checks by subsuming t3 to t1. The call is fine because the function e1 that evaluates to gets an argument with "more than what it needs." Contravariant subtyping is about subsuming some function itself to a different function type. For example, we can subsume t1->t2 to t3->t2, i.e., t3->t2 is a supertype of t1->t2 because t3 is a subtype of t2. This is a "different way" we could use subsumption to type-check e1(e2). For type-checking a call, either subsumption is fine, and certainly subsuming the argument is simpler to think about. But if we are going to have one object be a subtype of another or store a function in a record or pass a function to another function, etc., we may need function subtyping for this to type-check since are not "yet" actually calling the function. For example, we need function subtyping and contravariant arguments for this code:

```
t1->t2 x = ...
t3->t2 y = x
... maybe later x or y gets called ...
```