CSE341, Fall 2011, Lecture 25 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.

Types for Objects (in the Next Lecture)

We previously studied static types for functional programs, in particular ML's type system. ML uses its type system to prevent errors like treating a number as a function. A key source of expressiveness in ML's type system (not rejecting too many programs that do nothing wrong and programmers are likely to write) is *parametric polymorphism*, also known as *generics*.

So we should also study static types for object-oriented programs, such as those found in Java. If everything is an object (which is less true in Java than in Ruby), then the main thing we would want our type system to prevent is "method missing" errors, i.e., sending a message to an object that has no method for that message. If objects have fields accessible from outside the object (e.g., in Java), then we also want to prevent "field missing" errors. There are other possible errors as well, like calling a method with the wrong number of arguments.

While languages like Java and C# have generics these days, the source of type-system expressiveness most fundamental to object-oriented style is *subtype polymorphism*, also known as *subtyping*. ML does not have subtyping, though this decision is really one of language design (it would complicate type inference, for example).

Our plan for this lecture and the next one is to:

- Study subtyping
- Compare subtyping and generics, determining which idioms are best supported by each
- Combine subtyping and generics, showing that the result is even more useful than the sum of the two techniques

Subtyping for Records and Functions

It would be natural to study subtyping using Java since it is a well-known object-oriented language with a type system that has subtyping. But it is also fairly complicated, using classes and interfaces for types that describe objects with methods, overriding, static overloading, etc. While these features have pluses and minuses, they can complicate the fundamental ideas that underlie how subtyping should work.

So this lecture studies subtyping using only records (like in ML, things with named fields holding contents — basically objects with public fields, no methods, and no class names) and functions (like in ML or Racket). This will let us see how subtyping should — and should not — work, so that then the next lecture can then apply the results to the more complicated setting of a class-based object-oriented language.

This approach has the disadvantage that we cannot use any of the language we have studied: ML does not have subtyping and record fields are immutable, Racket and Ruby are dynamically typed, and Java is too complicated for our starting point. So we are going to make up a language with just records, functions, variables, numbers, strings, etc. and explain the meaning of expressions and types as we go.

A Made-Up Language of Records

To study the basic ideas behind subtyping, we will use records with mutable fields, as well as functions and other expressions. Our syntax will be a mix of ML and Java that keeps examples short and, hopefully, clear. For records, we will have expressions for making records, getting a field, and setting a field as follows:

- In the expression `{f1=e1, f2=e2, ..., fn=en}`, each `fi` is a field name and each `ei` is an expression. The semantics is to evaluate each `ei` to a value `vi` and the result is the record value `{f1=v1, f2=v2, ..., fn=vn}`. So a record value is just a collection of fields, where each field has a name and a contents.
- For the expression `e.f`, we evaluate `e` to a value `v`. If `v` is a record with an `f` field, then the result is the contents of the `f` field. Our type system will ensure `v` has an `f` field.
- For the expression `e1.f = e2`, we evaluate `e1` and `e2` to values `v1` and `v2`. If `v1` is a record with an `f` field, then we update the `f` field to have `v2` for its contents. Our type system will ensure `v1` has an `f` field. Like in Java, we will choose to have the result of `e1.f = e2` be `v2`, though usually we do not use the result of a field-update.

Now we need a type system, with a form of types for records and typing rules for each of our expressions. Like in ML, let's write record types as `{f1:t1, f2:t2, ..., fn:tn}`. For example, `{x : real, y : real}` would describe records with two fields named `x` and `y` that hold contents of type `real`. And `{foo: {x : real, y : real}, bar : string, baz : string}` would describe a record with three fields where the `foo` field holds a (nested) record of type `{x : real, y : real}`. We then type-check expressions as follows:

- If `e1` has type `t1`, `e2` has type `t2`, ..., `en` has type `tn`, then `{f1=e1, f2=e2, ..., fn=en}` has type `{f1:t1, f2:t2, ..., fn:tn}`.
- If `e` has a record type containing `f : t`, then `e.f` has type `t` (else `e.f` does not type-check).
- If `e1` has a record type containing `f : t` and `e2` has type `t`, then `e1.f = e2` has type `t` (else `e1.f = e2` does not type-check).

Assuming the "regular" typing rules for other expressions like variables, functions, arithmetic, and function calls, an example like this will type-check as we would expect:

```
fun distToOrigin (p:{x:real,y:real}) =
  Math.sqrt(p.x*p.x + p.y*p.y)

val pythag : {x:real,y:real} = {x=3.0, y=4.0}
val five : real = distToOrigin(pythag)
```

In particular, the function `distToOrigin` has type `{x : real, y : real} -> real`, where we write function types with the same syntax as in ML.

This type system does what it is intended to do: No program that type-checks would, when evaluated, try to look up a field in a record that does not have that field.

Now Add Subtyping

With our typing rules so far, this program would not type-check:

```
fun distToOrigin (p:{x:real,y:real}) =
  Math.sqrt(p.x*p.x + p.y*p.y)
val c : {x:real,y:real,color:string} = {x=3.0, y=4.0, color="green"}
val five : real = distToOrigin(c)
```

In the call `distToOrigin(c)`, the type of the argument is `{x:real,y:real,color:string}` and the type the function expects is `{x:real,y:real}`, breaking the typing rule that functions must be called with the

type of argument they expect. Yet the program above is safe: running it would not lead to accessing a field that does not exist.

A natural idea is to make our type system more lenient as follows: If some expression has a record type `{f1:t1, ..., fn:tn}`, then let the expression *also* have a type where some of the fields are removed. Then our example will type-check: Since the expression `c` has type `{x:real,y:real,color:string}`, it can also have type `{x:real,y:real}`, which allows the call to type-check. Notice we could also use `c` as an argument to a function of type `{color:string}->int`, for example.

Letting an expression that has one type also have another type that has less information is the idea of *subtyping*. (It may seem backwards that the *subtype* has *more* information, but that is how it works. A less-backwards way of thinking about it is that there are "*fewer*" values of the subtype than of the supertype because values of the subtype have more obligations, e.g., having more fields.)

We will now add subtyping to our made-up language, in a way that will not require us to change any of our existing typing rules. For example, we will leave the function-call rule the same, still requiring that the type of the actual argument *equal* the type of the function parameter in the function definition. To do this, we will add two things to our type system:

- The idea of one type being a subtype of another: We will write `t1 <: t2` to mean `t1` is a subtype of `t2`.
- One and only new typing rule: If `e` has type `t1` and `t1 <: t2`, then `e` (also) has type `t`.

So now we just need to give rules for `t1 <: t2`, i.e., when is one type a subtype of another.

Subtyping is Not a Matter of Opinion

A common misconception is that if we are defining our own language, then we can make the typing and subtyping rules whatever we want. That is only true if we forget that our type system is allegedly preventing something from happening when programs run. If our goal is (still) to prevent field-missing errors, then we cannot add any subtyping rules that would cause us to stop meeting our goal.

For subtyping, the key guiding principle is *substitutability*: If we allow `t1 <: t2`, then any value of type `t1` must be able to be used in every way a `t2` can be. For records, that means `t1` should have all the fields that `t2` has and with the same types.

Some Good Subtyping Rules

Without further ado, we can now give four rules that we can add to our language to accept more programs without breaking the type system. The first two are specific to records and the next two, while perhaps seeming unnecessary, do no harm and are common in any language with subtyping because they combine well with other rules:

- "Width" subtyping: A supertype can have a subset of fields with the same types, i.e., a subtype can have "extra" fields
- "Permutation" subtyping: A supertype can have the same set of fields with the same types in a different order.
- Transitivity: If `t1 <: t2` and `t2 <: t3`, then `t1 <: t3`.
- Reflexivity: Every type is a subtype of itself: `t <: t`.

Notice that width subtyping lets us forget fields, permutation subtyping lets us reorder fields (e.g., so we can pass a `{x:real,y:real}` in place of a `{y:real,x:real}`) and transitivity with those rules lets us do both (e.g., so we can pass a `{x:real,foo:string,y:real}` in place of a `{y:real,x:real}`).

Depth Subtyping: A Bad Idea With Mutation

Our subtyping rules so far let us drop fields or reorder them, but there is no way for a supertype to have a field with a different type than in the subtype. For example, consider this example, which passes a "sphere" to a function expecting a "circle." Notice that circles and spheres have a `center` field that itself holds a record.

```
fun circleY (c:{center:{x:real,y:real}, r:real}) =
   c.center.y

val sphere:{center:{x:real,y:real,z:real}, r:real}) = {center={x=3.0,y=4.0,z=0.0}, r=1.0}
val _ = circleY(sphere)
```

The type of `circleY` is `{center:{x:real,y:real}, r:real}->real` and the type of `sphere` is `{center:{x:real,y:real,z:real}, r:real}`, so the call `circleY(sphere)` can type-check only if

```
{center:{x:real,y:real,z:real}, r:real} <: {center:{x:real,y:real}, r:real}
```

This subtyping does not hold with our rules so far: We can drop the `center` field, drop the `r` field, or reorder those fields, but we cannot "reach into a field type to do subtyping."

Since we might like the program above to type-check since evaluating it does not nothing wrong, perhaps we should add another subtyping rule to handle this situation. The natural rule is "depth" subtyping for records:

- "Depth" subtyping: If `ta <: tb`, then `{f1:t1,...,f:ta,...,fn:tn} <: {f1:t1,...,f:tb,...,fn:tn}`.

This rule lets us use width subtyping on the field `center` to show

```
{center:{x:real,y:real,z:real}, r:real} <: {center:{x:real,y:real}, r:real}
```

so the program above now type-checks.

Unfortunately, this rule breaks our type system, allowing programs that we do not want to allow to type-check! This may not be intuitive and programmers make this sort of mistake often — thinking depth subtyping should be allowed. Here is an example:

```
fun setToOrigin (c:{center:{x:real,y:real}, r:real})=
   c.center = {x=0.0, y=0.0}

val sphere:{center:{x:real,y:real,z:real}, r:real}) = {center={x=3.0,y=4.0,z=0.0}, r=1.0}
val _ = setToOrigin(sphere)
val _ = sphere.center.z
```

This program type-checks in much the same way: The call `setToOrigin(sphere)` has an argument of type `{center:{x:real,y:real,z:real}, r:real}` and uses it as a `{center:{x:real,y:real}, r:real}`. But what happens when we run this program? `setToOrigin` mutates its argument so the `center` field holds a record *with no z field!* So the last line, `sphere.center.z` will not work: it tries to read a field that does not exist.

The moral of the story is simple if often forgotten: In a language with records (or objects) with getters and setters for fields, depth subtyping is unsound — you cannot have a different type for a field in the subtype and the supertype.

Note, however, that if a field is not settable (i.e., it is immutable), then the depth subtyping rule is sound and, like we saw with `circleY`, useful. So this is yet another example of how not having mutation makes programming easier. In this case, it allows more subtyping, which lets us reuse code more.

Another way to look at the issue is that given the three features of (1) setting a field, (2) letting depth subtyping change the type of a field, and (3) having a type system actually prevent field-missing errors, you can have any two of the three.

The Problem With Java/C# Array Subtyping

Now that we understand depth subtyping is unsound if record fields are mutable, we can question how Java and C# treat subtyping for arrays. For the purpose of subtyping, arrays are very much like records, just with field names that are numbers and all fields having the same type. (Since `e1[e2]` computes what index to access and the type system does not restrict what index might be the result, we need all fields to have the same type so that the type system knows the type of the result.) So it should very much surprise us that this code type-checks in Java:

```
class Point { ... } // has fields double x, y
class ColorPoint extends Point { ... } // adds field String color
...
void m1(Point[] pt_arr) {
  pt_arr[0] = new Point(3,4);
}
String m2(int x) {
  ColorPoint[] cpt_arr = new ColorPoint[x];
  for(int i=0; i < x; i++)
     cpt_arr[i] = new ColorPoint(0,0,"green");
  m1(cpt_arr);
  return cpt_arr[0].color;
}
```

The call `m1(cpt_arr)` uses subtyping with `ColorPoint[] <: Point[]`, which is essentially depth subtyping even though array indices are mutable. As a result, it appears that `cpt_arr[0].color` will read the `color` field of an object that does not have such a field.

What actually happens in Java and C# is the assignment `pt_arr[0] = new Point(3,4);` will raise an exception if `pt_arr` is actually an array of `ColorPoint`. In Java, this is an `ArrayStoreException`. The advantage of having the store raise an exception is that no other expressions, such as array reads or object-field reads, need run-time checks. The invariant is that an object of type `ColorPoint[]` always holds objects that have type `ColorPoint` or a subtype, not a supertype like `Point`. Since Java allows depth subtyping on arrays, it cannot maintain this invariant statically. Instead, it has a run-time check on all array assignments, using the "actual" type of array elements and the "actual" class of the value being assigned. So even though in the type system `pt_arr[0]` and `new Point(3,4)` both have type `Point`, this assignment can fail at run-time.

As usual, having run-time checks means the type system is preventing fewer errors, requiring more care and testing, plus the run-time cost of performing these checks on array updates. So why were Java and C# designed this way? It seemed important for flexibility before these languages had generics so that, for example, if you wrote a method to sort an array of `Point` objects, you could use your method to sort an array of `ColorPoint` objects. Allowing this makes the type system simpler and less "in your way" at the expense of statically checking less. Better solutions would be to use generics in combination with subtyping (see bounded polymorphism in the next lecture) or to have support for indicating that a method will not update array elements, in which case depth subtyping is sound.

`null` in Java/C#

While we are on the subject of pointing out places where Java/C# choose dynamic checking over the "natural" typing rules, the far more ubiquitous issue is how the constant `null` is handled. Since this value has no fields or methods (in fact, unlike `nil` in Ruby, it is not even an object), its type should naturally reflect that it cannot be used as the receiver for a method or for getting/setting a field. Instead, Java and C# allow `null` to have *any* object type, as though it defines *every* method and has *every* field. From a static checking perspective, this is exactly backwards. As a result, the language definition has to indicate that *every* field access and method call includes a run-time check for `null`, leading to the `NullPointerException` errors in Java you have surely encountered.

So why were Java and C# designed this way? Because there are situations where it is very convenient to have `null`, such as initializing a field of type `Foo` before you can create a `Foo` instance (e.g., if you are building a cyclic list). But it is also very common to have fields and variables that should never hold `null` and you would like to have help from the type-checker in maintaining this invariant. Many proposals for incorporating "can't be `null`" types into programming languages have been made, but none have yet "caught on" for Java or C#. In contrast, notice how ML uses option types for similar purposes: The types `t option` and `t` are not the same type; you have to use `NONE` and `SOME` constructors to build a datatype where values might or might not actually have a `t` value.

Function subtyping

The rules for when one function type is a subtype of another function typ are even less intuitive than the issue of depth subtyping for records, but they are just as important for understanding how to safely override methods in object-oriented languages (next lecture).

When we talk about function subtyping, we are talking about using a function of one type in place of a function of another type. For example, if `f` takes a function `g` of type `t1->t2`, can we pass a function of type `t3->t4` instead. If `t3->t4` is a subtype of `t1->t2` then this is allowed because, as usual, we can pass the function `f` an argument that is a subtype of the type expected. But this is not "function subtyping" on `f` — it is "regular" subtyping on function arguments. The "function subtyping" is deciding that one function type is a subtype of another.

To understand function subtyping, let's use this example of a higher-order function, which computes the distance between the two-dimensional point `p` and the result of calling `f` with `p`:

```
fun distMoved (f : {x:real,y:real}->{x:real,y:real},
               p : {x:real,y:real}) =
   let val p2 : {x:real,y:real} = f p
       val dx : real = p2.x - p.x
       val dy : real = p2.y - p.y
   in Math.sqrt(dx*dx + dy*dy) end
```

The type of `distMoved` is

```
(({x:real,y:real}->{x:real,y:real}) * {x:real,y:real}) -> real
```

So a call to `distMoved` requiring no subtyping could look like this:

```
fun flip p = {x = ~p.x, y=~p.y}
val d = distMoved(flip, {x=3.0, y=4.0})
```

The call above could also pass in a record with extra fields, such as `{x=3.0,y=4.0,color="green"}`, but this is just ordinary width subtyping on the second argument to `distMoved`. Our interest here is deciding what

functions with types other than `{x:real,y:real}->{x:real,y:real}` can be passed for the first argument to `distMoved`.

First, it is safe to pass in a function with a return type that "promises" more, i.e., returns a subtype of the needed return type for the function `{x:real,y:real}`. For example, it is fine for this call to type-check:

```
fun flipGreen p = {x = ~p.x, y=~p.y, color="green"}
val d = distMoved(flipGreen, {x=3.0, y=4.0})
```

The type of `flipGreen` is

```
{x:real,y:real} -> {x:real,y:real,color:string}
```

This is safe because `distMoved` expects a `{x:real,y:real}->{x:real,y:real}` and `flipGreen` is substitutable for values of such a type since the fact that `flipGreen` returns a record that also has a `color` field is not a problem.

In general, the rule here is that if `ta <: tb`, then `t -> ta <: t -> tb`, i.e., the subtype can have a return type that is a subtype of the supertype's return type. To introduce a little bit of jargon, we say return types are *covariant* for function subtyping meaning the subtyping for the return types works "the same way" (co) as for the types overall.

Now let us consider passing in a function with a different argument type. It turns out argument types are NOT covariant for function subtyping. Consider this example call to `distMoved`:

```
fun flipIfGreen p = if p.color = "green"
                    then {x = ~p.x, y=~p.y}
                    else {x = p.x, y=p.y}
val d = distMoved(flipIfGreen, {x=3.0, y=4.0})
```

The type of `flipIfGreen` is

```
{x:real,y:real,color:string} -> {x:real,y:real}
```

This program should not type-check: If we run it, the expression `p.color` will have a "no such field" error since the point passed to `flipIfGreen` does not have a `color` field. In short, `ta <: tb`, does NOT mean `ta -> t <: tb -> t`. This would amount to using a function that "needs more of its argument" in place of a function that "needs less of its argument." This breaks the type system since the typing rules will not require the "more stuff" to be provided.

But it turns out it works just fine to use a function that "needs less of its argument" in place of a function that "needs more of its argument." Consider this example use of `distMoved`:

```
fun flipX_Y0 p = {x = ~p.x, y=0.0}
val d = distMoved(flipX_Y0, {x=3.0, y=4.0})
```

The type of `flipX_Y0` is

```
{x:real} -> {x:real,y:real}
```

since the only field the argument to `flipX_Y0` needs is `x`. And the call to `distMoved` causes no problem: `distMoved` will always call its `f` argument with a record that has an `x` field and a `y` field, which is more than `flipX_Y0` needs.

In general, the treatment of argument types for function subtyping is "backwards" as follows: If `tb <: ta`, then `ta -> t <: tb -> t`. The technical jargon for "backwards" is *contravariance*, meaning the subtyping for argument types is the reverse (contra) of the subtyping for the types overall.

As a final example, function subtyping can allow contravariance of arguments and covariance of results:

```
fun flipXMakeGreen p = {x = ~p.x, y=0.0, color="green"}
val d = distMoved(flipXMakeGreen, {x=3.0, y=4.0})
```

Here `flipXMakeGreen` has type

```
{x:real} -> {x:real,y:real,color:string}
```

This is a subtype of

```
{x:real,y:real} -> {x:real,y:real}
```

because `{x:real,y:real} <: {x:real}` (contravariance on arguments) and
`{x:real,y:real,color:string} <: {x:real,y:real}` (covariance on results).

The general rule for function subtyping is: If `t3 <: t1` and `t2 <: t4`, then `t1->t2 <: t3->t4`. This rule, combined with reflexivity (every type is a subtype of itself) lets us use contravariant arguments, covariant results, or both.

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 very smart people get confused because it is *not* about calls to methods/functions. Rather it is about the methods/functions themselves. We do not need function subtyping for passing non-function arguments to functions: we can just use other subtyping rules (e.g., those for records). Function subtyping is needed for higher-order functions or for storing functions themselves in records. And object types are related to having records with functions (methods) in them.