Subtyping

Basics of subtyping

The concept of subtyping is particularly critical in most object-oriented programming languages, but it is also useful in other settings.

> We say that a type `t1` is a subtype of another type `t2`, and we write `t1 <: t2`, if a value of type `t1` can be used anywhere where a value of type `t2` is expected.

A good example was our discussion of 2-dimensional points vs 3-dimensional points. Any operation that expects a 2-dimensional point will make use of the two coordinates `x` and `y`. But a 3-dimensional point also has those two coordinates. So should we allow these operations to be performed on 3-dimensional points as well? i.e. is `point3d` a subtype of `point2d`? This is a crucial question for object-oriented languages: Classes are often identified with types in that case, and a subclass relation must also correspond to a subtype relation, as the main point of subclasses is that their objects can be used wherever objects of the superclass could.

The above example points out that records are at a relevant topic of discussion for subtyping. We can consider a record type to be a subtype of another if:

- It has all the same fields with the same types, but possibly has extra fields. This is called *width subtyping*.
- It has all the same fields, but possibly with subtypes of the corresponding types on each field. This is called *depth subtyping*.

It is an important question to decide what kinds of rules subtyping should obey. We can list some of these rules:

- Every type should be its own subtype: `t <: t` (reflexive property)
- A subtype of a type's subtype is also a subtype of the larger type (transitivity): If `t1 <: t2` and `t2 <: t3` then `t1 <: t3`.
- Width subtyping should be valid: `{a: t1, b: t2, c: t3} <: {a: t1, b: t2}`
- Permutation subtyping: The order of fields in width subtyping should not matter (or in other words, rearranging the order of the fields produces a supertype of our type).

Depth subtyping and mutation

The presence of mutation poses some interesting problems with depth subtyping. We might want a rule that says:

- If `t1 <: t2`, then `{a: t1, ...} <: {a: t2, ...}` where all other fields are kept the same.

But if we are allowed to mutate a record's fields, we might have the following series of operations:

```
let change_center (c: {center: {x: x}, r: r}) =
  c.center := {x: 0};;

let sph = {center= { x= 0, y= 0}, r= 3};;
change_center sph;;
sph.center.y;;        <---------- Will get an error!
```

This is a series of operations the typechecker should allow:

- change_center takes as input a value of type {center: {x: int}, r: int}. It replaces the value of center with a value of type {x: int}.
- sph starts off as a value of type {center: {x: int, y: int}, r: int}.
- If depth subtyping were to be a valid rule, then sph would be a valid input to the change_center function.
- Since sph has type {center: {x: int, y: int}, r: int}, we should be allowed to access sph.center.y.

And yet at the end of the day we end up accessing an entry that is not there! Our type system is no longer sound! This is the kind of error it was supposed to protect us from. So we cannot have mutable record fields and depth subtyping in a sound type system.

Arrays and subtyping

A similar problem arises in the use of (mutable) arrays in languages like Java. Consider for example the following code, borrowed from Dan Grossman's course:

```
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;
}
```

Let's describe what is going on:

- We have a class Point, and subclass ColorPoint. This also establishes a subtype relation.

- We have a function m1 that takes an array of Point objects, and changes the 0-th entry in that array to a new Point.
- We have a second function m2 that takes an integer, then creates an array of that many ColorPoint instances.
- This array can be given as input to the first function m1 that was expecting a Point array, as ColorPoint is a subtype of Point. This is essentially a form of depth subtyping.
- That function m1 changes the first entry in the array (to an instance of Point).
- We then try to access the color field of the first entry in the array. At this point that entry is an instance of Point, not ColorPoint, so we would get an error.

Clearly something went very wrong. The question is what, and how to prevent it. What should we "fix"?

Java's approach is as follows: When m1 runs, even though on a static typing level it expected a Point array, it has runtime information that it is indeed a ColorPoint array. So when m1 tries to put a Point instance in the first entry, there will be a runtime exception thrown because Point is not a subtype of ColorPoint and the array is actually a ColorPoint array.

So in essence, in order to allow both depth subtyping in the form of arrays and mutation, Java is required to insert some runtime checks instead of static checks to ensure that when a value is mutated then a value of the correct runtime "type" is inserted. But it does mean that our static type system is catching fewer errors.

If we have time we will discuss *bounded polymorphism*, which affords a way to control such problems at the static level.

Function subtyping

In this section we address a simple question: When is a function type T1 –> S1 a subtype of another function type T2 –> S2?

For this we apply the essence of subtyping: We should be able to use the function f of type T1 –> S1 anywhere where a function of type T2 –> S2 is expected. Let us see what this means:

- We should be able to think of f as a function of type T2 –> S2.
- So we must be able to call it with an input of type T2. But since f can only handle inputs of type T1, it must be the case that T2 <: T1.
- The result of such a call would be of type S1. But we must be able to think of it as a result of type S2. So it must be the case that S1 <: S2.

So we have:

> We have the function subtype relation (T1 –> S1) <: (T2 –> S2) if and only if T2 <: T1 and S1 <: S2.

We say that function types are **covariant** in the return type but **contravariant** in the argument type.

Generics plus subtyping: The need for bounded polymorphism

Both generics (polymorphic types) and subtyping are desirable features, and while including them both in a language design is doable, it presents some questions, especially in relation to mutation.

As an example, suppose we have a structure for mutable lists, List<T>. And suppose we have two types, ColorPoint <: Point. Then we would like to write a function that "filters" those points that are within say a given distance from the origin. So a function:

```
closeEnough: List<Point> –> List<Point>
```

Suppose further that we have a list of ColorPoints:

```
List<ColorPoint> lst = [p1, p2, p3]
```

Then we would like to feed that into the closeEnough function:

```
lst2 = closeEnough lst
```

The first problem we encounter is whether we should be allowed to do this at all. After all closeEnough expected a List<Point> not a List<ColorPoint>. If we want our lists to be mutable, we have the depth subtyping problem we discussed earlier, and we cannot say that List<ColorPoint> <: List<Point>.

The other problem we have is that of the return value. The definition of closeEnough tells us merely that this is a List<Point>. But in reality it is a List<ColorPoint> as we merely filtered the original list. We have however lost that fact.

So you might say, why can't we just make closeEnough fully polymorphic:

```
closeEnough: List<T> –> List<T>
```

Then we could ensure that whatever we get back has the same type as what went in. The problem is that the computation that closeEnough needs to perform requires something of the provided element, namely that it be a Point so we can measure its distance. This is not possible for a generic type.

What we need is in some sense the best of both worlds: We need a polymorphic type, but we need it to be restricted to only those types that are subtypes of Point. This is **bounded polymorphism**, where we do have a variable type but in a somewhat restricted way, bounded to be a subtype of a specific type.

With that in mind, the type of closeEnough may look like this:

```
<T extends Point> closeEnough: List<T> –> List<T>
```

We can only apply closeEnough to a type that is a subtype of Point. Then closeEnough typechecks because it can count on its elements being subtypes of Point and therefore having the necessary distance properties. And also when we call it with ColorPoint it will then take the place of T and we can guarantee that the return type is List<T>, i.e. List<ColorPoint>.