The last couple lectures have covered *subtype polymorphism*, also known as *subtyping*. Earlier in the course we learned about *parametric polymorphism*, also known as *generic types*, or just *generics*. This lecture compares and contrasts the two approaches, demonstrates what each is designed for, and considers using both together via *bounded polymorphism*.

1 Generics

There are many programming idioms that use generic types. We do not consider all of them here, but let’s reconsider probably the two most common idioms that came up when studying higher-order functions.

First, there are functions that combine other functions such as `compose`:

```plaintext
val compose : ('b -> 'c) * ('a -> 'b) -> ('a -> 'c)
```

Second, there are functions that operate over collections where different collections can hold values of different types:

```plaintext
val length : 'a list -> int
val map : ('a -> 'b) -> 'a list -> 'b list
val swap : ('a * 'b) -> ('b * 'a)
```

In all these cases, the key point is that if we had to pick non-generic types for these functions, we would get significantly less code reuse. For example, we would need one `swap` function for producing an `int * bool` from a `bool * int` and another `swap` function for swapping the positions of an `int * int`.

Generic types are much more useful and precise than just saying that some argument can “be anything.” For example, the type of `swap` indicates that the second component of the result has the same type as the first component of the argument and the first component of the result has the same type as the second component of the argument. Type variables indicate when multiple values can have any type but must be the same type. Recall, different type variables can be instantiated with the same type or with different types.

**Generics in Java.** Java has had subtype polymorphism since its creation in the 1990s and has had parametric polymorphism since 2004. Using generics in Java can be more cumbersome than in ML, but generics are still useful for the same programming idioms. Here, for example, is a generic `Pair` class, allowing the two fields to have any type:

```java
class Pair<T1,T2> { 
    T1 x;
    T2 y;
    Pair(T1 x, T2 y) { this.x = x; this.y = y; }
    Pair<T2,T1> swap() { return new Pair<T2,T1>(y,x); }
}
```

As in ML, `Pair` should not be a type, but rather should be a type constructor. (In truth, Java does let you use it as a type, and we’ll see later in this lecture that creates serious problems.) `Pair` can be instantiated (e.g., `Pair<String,Integer>`) to create a type. The `swap` method is, in object-oriented style, an instance method in `Pair<T1,T2>` that returns a `Pair<T2,T1>`. We could also define a static method:
static <T1,T2> Pair<T2,T1> swap(Pair<T1,T2> p) {
    return new Pair<T2,T1>(p.y,p.x);
}

Java doesn’t have a Pair class in the the standard library. The Android API does (android.util.Pair).

2 Subtyping is a poor substitute for generics

If a language does not have generics, or if a programmer is not comfortable with them, one often sees code that should use generics written in terms of subtyping instead. Doing so is like painting with a hammer instead of a paintbrush: technically possible, but clearly the wrong tool. Consider this Java example:

class LamePair {
    Object x;
    Object y;
    LamePair(Object x, Object y){ this.x=x; this.y=y; }
    LamePair swap() { return new LamePair(y,x); }
}
String s = (String)(new LamePair("hi",4).y); // error caught only at run-time

The code in LamePair type-checks without problem: the fields x and y have type Object, which is a supertype of every class and interface type. The difficulties arise when clients use this class. Passing arguments to the constructor works as expected with subtyping (Java will even automatically convert int literal 4 to an Integer object holding 4). But when we retrieve the contents of a field, an Object is not very useful: we’d rather the type system knew that the type of the value put in is the type of the value retrieved.

Unfortunately, subtyping does not work that way: the type system knows only that the field holds an Object. So we have to use a downcast, e.g., (String) e, which is a run-time check that the result of evaluating e is actually of type String or a subtype thereof. Such run-time checks have the usual dynamic-checking costs in terms of performance, but, more importantly, in terms of the possibility of failure: they can’t be checked statically. Indeed, in the example above, the downcast would fail: it is the x field that holds a String, not the y field.

When you use Object and downcasts, as in the code above, you are essentially taking a dynamic typing approach. Any object could be stored in an Object field, so it is up to you, without help from the type system, to keep straight what types of objects are stored where.

Subtyping is useful. We do not mean to suggest that subtyping is useless. In fact, it’s great for allowing code to be reused with data that has “extra information.” For example, geometry code that operates over type Point should work fine type ColorPoint. In general, subtyping works well with GUI libraries. Code in such libraries typically works fine for any sort of graphical element (“paint it on the screen,” “change the background color,” “report if the mouse is clicked on it,” etc.). Different elements such as buttons, slider bars, or text boxes can then be subtypes of some common supertype.

The lack of subtyping in ML can be annoying. It is certainly inconvenient in such situations that ML code like this simply does not type-check:

fun distToOrigin1 {x=x,y=y} = 
    Math.sqrt (x*x + y*y)

(* does not type-check *)
(* val five = distToOrigin1 {x=3.0,y=4.0,color="red"} *)
3 Generics are a poor substitute for subtyping

In a language like ML, which has generics but not subtyping, you can still “code up” something similar with higher-order functions. But it’s a lot of work to express a simple idea. For example, distToOrigin2 below uses getters passed in by the caller to access the x and y fields. The next two functions have identical bodies, but must have different types just to appease the type-checker.

```ml
fun distToOrigin2(getx, gety, v) =  
  let
    val x = getx v
    val y = gety v
  in
    Math.sqrt (x*x + y*y)
  end

fun distToOriginPt (p : {x:real,y:real})  =
  distToOrigin2(fn v => #x v,  
                 fn v => #y v,  
                 p)

fun distToOriginColorPt (p : {x:real,y:real,color:string}) = 
  distToOrigin2(fn v => #x v,  
                 fn v => #y v,  
                 p)
```

4 Bounded polymorphism

As Java and C# demonstrate, a language can have both generic types and subtyping. In addition to the obvious benefit of supporting the idioms peculiar to both features, the two can be combined to yield even more code reuse and expressiveness.

When combined, generics and subtypes yield bounded polymorphism: instead of expressing “a subtype of T” or “for all types ‘a,” we can express something richer, “for all types ‘a that are a subtype of T.” As with generics, ‘a can be used multiple times in a type to indicate where two part of that type must have the same type. As with subtyping, ‘a is a subtype of T, so code can safely access whatever fields and methods T has.

Consider this Point class with a distance method:

```java
class Pt {
  double x, y;
  Pt(double x, double y) { this.x = x; this.y = y; }
  double distance(Pt pt) {
    double dx = x-pt.x;
    double dy = y-pt.y;
    return Math.sqrt(dx*dx+dy*dy);
  }
}
```

Suppose we want to write a method that takes a list of points pts, a point center, and a radius radius and returns a new list of points containing all the input points within radius of center, i.e., within the circle defined by center and radius. We’ll now embark on an extended example to see why bounded polymorphism is needed to write such a method.
Generics alone. Here’s a first attempt to write the method using generics alone:

```java
static List<Pt> inCircle1(List<Pt> pts, Pt center, double radius) {
    List<Pt> result = new LinkedList<Pt>();
    for (Pt pt : pts)
        if (pt.distance(center) <= radius)
            result.add(pt);
    return result;
}
```

This code works perfectly fine for a `List<Pt>` argument. But suppose `ColorPt` is a subtype of `Pt` (adding a color field and associated methods). Because depth subtyping is unsound with mutable fields, `List<ColorPt>` is not a subtype of `List<Pt>`. So we cannot call `inCircle1` with a `List<ColorPt>` argument:

```java
List<ColorPt> cps = ...
Pt p = ...
List<ColorPt> out0 = inCircle1(cps, p, 1.5); // does not type-check
```

There are two reasons the above code won’t type-check. First, `cps` is a `List<ColorPt>`, not a `List<Pt>`. So the call to `inCircle1` won’t type-check. Second, even though at run-time `inCircle1` will return a `List<ColorPt>` when the argument is a `List<ColorPt>`, the type of `inCircle1` says that it always returns a `List<Pt>`. So the assignment to `out0` doesn’t type-check.

Generics plus run-time casts. To solve the problem with the previous code not type-checking, let’s now consider inserting run-time casts. Here’s what you might expect would work:

```java
List<ColorPt> out0 = (List<ColorPt>) inCircle1((List<Pt>) cps, p, 1.5);
```

But that doesn’t type-check either, for reasons having to do with backwards compatibility with Java before 5 (which is when generics were added). The problem is that the run-time actually “forgets” what `T` is in any generic type like `List<T>`. At run-time, it’s possible to determine (e.g.) that an object is a `String` rather than a `Integer`, but it’s not possible to determine whether an object is a `List<String>` or a `List<Integer>`. So it’s impossible for the run-time cast to check whether `cps` is a `List<Pt>`, and the compiler therefore rejects the code.

There is a way to get around this “feature,” but it’s not pretty. We have to use the type constructor `List` to do it:

```java
List<ColorPt> out1 = (List<ColorPt>)(List) inCircle1((List<Pt>)(List)cps, p, 1.5);
```

Here, Java lets us use a type constructor as if it were really a type. (Java calls `List` used in this way a “raw type.”) The code now compiles, but with warnings:

Note: lec23.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.

Recompiling with that flag yields warnings about unchecked casts. The compiler is warning us that the run-time will not be able to check whether the casts are sound. **Unchecked casts are dangerous.**

For `inCircle1`, in turns out that the casts are safe: if `inCircle` is passed a `List<ColorPt>` the result will be a `List<ColorPt>`. But consider this variant of `inCircle1`:
static List<Pt> inCircle2(List<Pt> pts, Pt center, double radius) {
    List<Pt> result = new LinkedList<Pt>();
    for (Pt pt : pts)
        if (pt.distance(center) <= radius)
            result.add(pt);
        else
            result.add(center);
    return result;
}

The difference is that any points not within the circle are “replaced” in the output by center. This call will type-check:

List<ColorPt> out2 = (List<ColorPt>)(List) inCircle2((List<Pt>)(List) cps, p, 1.5)

But if List<ColorPt> cps contains a point that isn’t within the circle, then the result should not be a List<ColorPt>, because it contains a Pt object (the center). So you might expect that the cast of the result to List<ColorPt> would fail at run-time, but it succeeds. So now we have a List<ColorPt> that isn’t actually a list of ColorPt objects. Later, when we try to extract the elements of the list, we’ll try to use a Pt as a ColorPt, and that cast will fail. The blame is clearly in the wrong place. So generics plus run-time casts don’t solve our problem—they just create new problems.

**Bounded polymorphism.** Bounded polymorphism solves our problems by combining subtyping with generics:

```java
static <T extends Pt> List<T> inCircle3(List<T> pts, Pt center, double radius) {
    List<T> result = new LinkedList<T>();
    for (T pt : pts)
        if (pt.distance(center) <= radius)
            result.add(pt);
    return result;
}
```

Annotation `<T extends Pt>` means that T is a type that is in scope in the method, and that T <: Pt. The compiler will check to make sure the method is never invoked with a type T that isn’t a subtype of Pt. Subtyping is necessary so that the method body can call the distance method on objects of type T. This code expresses exactly what we want! No run-time casts are necessary, and no run-time errors can occur.

We could debate what type is best for the center argument to inCircle3. Above, we chose Pt, but we could also choose T:

```java
static <T extends Pt> List<T> inCircle3alt(List<T> pts, T center, double radius) {
    List<T> result = new LinkedList<T>();
    for (T pt : pts)
        if (pt.distance(center) <= radius)
            result.add(pt);
    return result;
}
```

It turns out this version allows fewer callers than the previous version allows. For example, inCircle3 allows a first argument of type List<ColorPt> and a second argument of type Pt. But inCircle3alt requires a second argument of type ColorPt when the first argument has type List<ColorPt>. 

5
On the other hand, adapting `inCircle2` (which, recall, sometimes adds `center` to the output) to use bounded polymorphism requires us to give the second argument type `T`:

```java
static <T extends Pt> List<T> inCircle4(List<T> pts, T center, double radius) {
    List<T> result = new LinkedList<T>();
    for (T pt : pts)
        if (pt.distance(center) <= radius)
            result.add(pt);
        else
            result.add(center);
    return result;
}
```

The reason we have to use `T` as the type of `center` is the call `result.add(center)`. That call wouldn't type-check if `center` had type `Pt`. (The actual compiler error that would result might be a bit confusing: it reports there is no `add` method for `List<T>` that takes a `Pt`.)