CSE341, Fall 2011, Lecture 27 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.

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*.

What are generics good for?

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`:

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

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

```
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 end up with 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. In general, we reuse a type variable to indicate when multiple things can have any type but must be the same type. Also recall that different type variables may or may not be instantiated with the same type.

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 without ML's support for type inference and, as a separate matter, closures, 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:

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

Notice that, analogous to ML, "`Pair`" is not a type: something like `Pair<String,Integer>` is 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);
}
```

For reasons of backwards-compatibility, the previous paragraph is not quite true: Java also has a type `Pair` that "forgets" what the types of its fields are. Casting to and from this "raw" type leads to compile-time warnings that you would be wise to heed: Ignoring them can lead to run-time errors in places you would not expect. The example below with bounded polymorphism has an example.

Subtyping is a Bad Substitute for Generics

If a language does not have generics or a programmer is not comfortable with them, one often sees generic code 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){ x=_x; 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. The difficulties arise when clients use this class. Passing arguments to the constructor works as expected with subtyping (plus, as in the example, Java will automatically convert a `4` to an `Integer` object holding a `4`). But when we retrieve the contents of a field, getting an `Object` is not very useful: we want the type of value we put back in.

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, in general, 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: this is not checked statically. Indeed, in the example above, the downcast would fail: it is the `x` field that holds a `String`, not the `y` field.

In general, when you use `Object` and downcasts, you are essentially taking a dynamic typing approach: any object could be stored in an `Object` field, so it is up to programmers, without help from the type system, to keep straight what kind data is where.

What is Subtyping Good For?

We do not suggest that subtyping is not useful: It is great for allowing code to be reused with data that has "extra information." For example, geometry code that operates over points should work fine for colored-points. 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"} *)
```

A generally agreed upon example where subtyping works well is graphical user interfaces. Much of the code for graphics libraries 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.) where different elements such as buttons, slider bars, or text boxes can then be subtypes.

Generics are a Bad Substitute for Subtyping

In a language with generics instead of subtyping, you can code up your own code reuse with higher-order functions, but it can be quite a bit of trouble for a simple idea. For example, `distToOrigin2` below uses getters passed in by the caller to access the `x` and `y` fields and then the next two functions have different types but identical bodies, just to appease the type-checker.

```
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)
```

Nonetheless, without subtyping, it may sometimes be worth writing code like `distToOrigin2` if you want it to be more reusable.

Bounded Polymorphism: Getting the Benefits of Both

As Java and C# demonstrate, there is no reason why a statically typed programming language cannot have generic types and subtyping. There are some complications from having both that we will not discuss (e.g., static overloading and subtyping are more difficult to define), but there are also benefits. In addition to the obvious benefit of supporting separately the idioms that each feature supports well, we can combine the ideas to get even more code reuse and expressiveness.

The key idea is to have *bounded generic types*, where instead of just saying "a subtype of `T`" or "for all types `'a`," we can say, "for all types `'a` that are a subtype of `T`." Like with generics, we can then use `'a` multiple times to indicate where two things must have the same type. Like with subtyping, we can treat `'a` as a subtype of `T`, accessing whatever fields and methods we know a `T` has.

An Extended Example of Bounded Polymorphism in Java

(For the full code for this example, see the Java file associated with this example.)

Consider this `Point` class with a `distance` method:

```
class Pt {
    double x, y;
    double distance(Pt pt) { return Math.sqrt((x-pt.x)*(x-pt.x)+(y-pt.y)*(y-pt.y));  }
    Pt(double _x, double _y) { x = _x; y = _y; }
}
```

Now consider this static 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`:

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

This code works perfectly fine for a `List<Pt>`, but if `ColorPt` is a subtype of `Pt` (adding a `color` field and associated methods), then we cannot call `inCircle` method above with a `List<ColorPt>` argument. Because depth subtyping is unsound with mutable fields, `List<ColorPt>` is not a subtype of `List<Pt>`. Even if it were, we would like to have a result type of `List<ColorPt>` when the argument type is `List<ColorPt>`.

For the code above, this is true: If the argument is a `List<ColorPt>`, then the result will be too, but we want a way to express that in the type system. Java's bounded polymorphism lets us describe this situation:

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

This method is polymorphic in type `T`, but `T` must be a subtype of `Pt`. This subtyping is necessary so that the method body can call the `distance` method on objects of type `T`. Wonderful!

While this second version is ideal, let us now consider a few variations. First, Java does have enough dynamically checked casts that it is possible to use the first version with a `List<ColorPt>` argument and cast the result from `List<Pt>` to `List<ColorPt>`. We have to use the "raw type" `List` to do it, something like this where `cps` has type `List<ColorPt>`.

```
List<ColorPt> out = (List<ColorPt>)(List) inCircle((List<Pt>)(List)cps, new Pt(0.0,0.0), 1.5);
```

In this case, these casts turn out to be okay: if `inCircle` is passed a `List<ColorPt>` the result will be a `List<ColorPt>`. But casts like this are dangerous. Consider this variant of the method that has the same type as the initial non-generic `inCircle` method:

```
static List<Pt> inCircle(List<Pt> pts, Pt center, double radius) {
    List<Pt> result = new ArrayList<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`. Now if we call `inCircle` with a `List<ColorPt> cps` where one of the points is not within the circle, then the result is *not* a `List<ColorPt>` — it contains a `Pt` object! You might expect then that the cast of the result to `List<ColorPt>` would fail, but Java does not work this way for backward-compatibility reasons: even this cast succeeds. So now we have a value of type `List<ColorPt>` that is not a list of `ColorPt` objects. What happens instead in Java is that a cast will fail later when we get a value from this alleged `List<ColorPt>` and try to use it as as `ColorPt` when it is in fact a `Pt`. The blame is clearly in the wrong place, which is why using the warning-inducing casts in the first past is so problematic.

Last, we can discuss what type is best for the `center` argument in our bounded-polymorphic version. Above, we chose `Pt`, but we could also choose `T`:

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

It turns out this version allows *fewer* callers since the previous version allows, for example, a first argument of type `List<ColorPt>` and a second argument of type `Pt` (and, therefore, via subtyping, also a `ColorPt`). With the argument of type `T`, we require a `ColorPt` (or a subtype) when the first argument has type `List<ColorPt>`. On the other hand, our version that sometimes adds `center` to the output requires the argument to have type `T`:

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

In this last version, if `center` has type `Pt`, then the call `result.add(center)` does not type-check since `Pt` may not be a subtype of `T` (what we know is `T` is a subtype of `Pt`). The actual error message may be a bit confusing: It reports there is no `add` method for `List<T>` that takes a `Pt`, which is true: the `add` method we are trying to use takes a `T`.