CSE341: Programming Languages

Lecture 27
Generics vs. Subtyping; Bounded Polymorphism

Dan Grossman

Fall 2011

Today

1. Compare generics and subtyping
   - What each is good for

2. Combine generics and subtyping to get even more benefit
   - Example in Java, but as always, ideas more general

What are generics good for?

Some good uses for parametric polymorphism:

- Types for functions that combine other functions:

```
fun compose (g,h) = fn x => g (h x)
(* compose : ('b -> 'c) * ('a -> 'b) -> ('a -> 'c) *)
```

- Types for functions that operate over generic collections

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

- Many other idioms
- General point: When types can "be anything" but multiple things need to be "the same type"

Generics in Java

- Java generics a bit clumsier syntactically and semantically, but can express the same ideas
  - Without closures, often need to use (one-method) objects
  - See also lecture on closures in Java/C
- Simple example without higher-order functions:

```
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);
  }
  …
}
```

Subtyping is not good for this

- Using subtyping for containers is much more painful for clients
  - Have to downcast items retrieved from containers
  - Downcasting has run-time cost
  - Downcasting can fail: no static check that container has the type of data you think it does
  - (Only gets more painful with higher-order functions like **`map`**)

```
class LamePair {
  Object x;
  Object y;
  LamePair(Object _x, Object _y){ x=_x; y=_y; }
  LamePair swap() { return new LamePair(y,x); }
}

// error caught only at run-time:
String s = (String)(new LamePair("hi",4).y);
```

What is subtyping good for?

Some good uses for subtype polymorphism:

- Code that "needs a Foo" but fine to have "more than a Foo"
  - Geometry on points works fine for colored points
  - GUI widgets specialize the basic idea of "being on the screen" and "responding to user actions"

- Related perspective: Writing code in terms of what it expects of arguments (but more is fine)
  - Static checking makes sure arguments have what is needed

Awkward in ML

ML does not have subtyping, so this simply does not type-check:

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

val five = distToOrigin {x=3.0,y=4.0,color="red"}
```

Higher-order workaround

- Can write reusable code in ML a la subtyping if you plan ahead and use generics in awkward ways

- See example in **`lec27.sml`**

Wanting both

- Could a language have generics and subtyping?
  - Sure!

- More interestingly, want to combine them
  - "Any type **`T1`** that is a subtype of **`T2`**"
  - This is bounded polymorphism
  - Lets you do things naturally you can't do with generics or subtyping

Example [also see Lec27.java]

- Only bounded polymorphism lets us use **`inCircle`** with a list of **`ColorPt`** objects
  - And callee can't put a **`Pt`** in **`pts`** or the result list!

```
class Pt {
  …
  double distance(Pt p) { … }
}
class ColorPt extends Pt { … }

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

One caveat

- For backward-compatibility and implementation reasons, in Java there is always a way to use casts to get around the static checking with generics
  - With or without bounded polymorphism

- Oh well