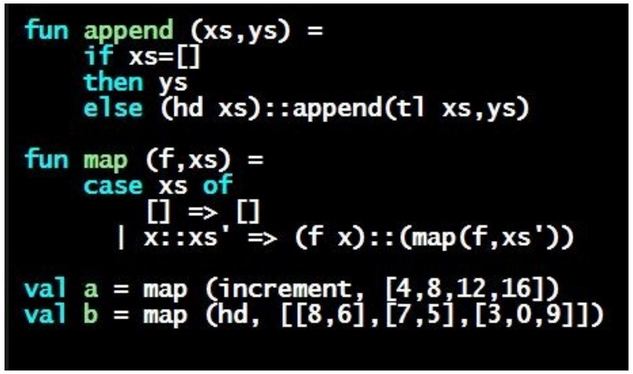

```
fun append (xs,ys) =
    if xs=[]
    then ys
    else (hd xs)::append(tl xs,ys)

fun map (f,xs) =
    case xs of
        [] => []
      | x::xs' => (f x)::(map(f,xs'))

val a = map (increment, [4,8,12,16])
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman
2013

Bounded Polymorphism

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 cannot do with generics or subtyping separately

Example

Method that takes a list of points and a circle (center point, radius)

- Return new list of points in argument list that lie within circle

Basic method signature:

```
List<Point> inCircle(List<Point> pts,
                     Point center,
                     double r) { … }
```

Optional: Java implementation straightforward assuming **`Point`** has a **`distance`** method

```
List<Point> result = new ArrayList<Point>();
for(Point pt: pts)
  if(pt.distance(center) <= r)
    result.add(pt);
return result;
```

Subtyping?

```
List<Point> inCircle(List<Point> pts,
                     Point center,
                     double r) { … }
```

- Would like to use **`inCircle`** by passing a **`List<ColorPoint>`** and getting back a **`List<ColorPoint>`**

- Java rightly disallows this: While **`inCircle`** would "do nothing wrong" its type does not prevent:
  - Returning a list that has a non-color-point in it
  - Modifying **`pts`** by adding non-color-points to it

Generics?

```
List<Point> inCircle(List<Point> pts,
                     Point center,
                     double r) { … }
```

- We could change the method to be

```
List<T> inCircle(List<T> pts,
                 Point center,
                 double r) { … }
```

  - Now the type system allows passing in a **`List<Point>`** to get a **`List<Point>`** returned or a **`List<ColorPoint>`** to get a **`List<ColorPoint>`** returned
  - But we cannot implement **`inCircle`** properly because method body should have no knowledge of type **`T`**

Bounds

- What we want:

```
List<T> inCircle(List<T> pts,
                 Point center,
                 double r) where T <: Point
 { … }
```

- Caller uses it generically, but must instantiate **`T`** with a subtype of **`Point`** (including **`Point`**)
- Callee can assume **`T <: Point`** so it can do its job
- Callee must return a **`List<T>`** so output will contain only list elements from input

Optional: *Real Java*

- The actual Java syntax

```
<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;
}
```

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