Polymorphisms: subtype vs. parametric

Can a type system support both ML-style type parameters and OO-style subtype rules?

Uses slides by Steve Freund

Java 1.0 (Subtype Poly.) vs. Java 1.5+ (Parametric Poly.)

```
class Stack {
  void push(Object o) {...}
  Object pop() {...}
  ...
}

String s = "Hello";
Stack st = new Stack();

st.push(s);

String t = (String)st.pop();
```

```
class Stack<T> {
  void push(T o) {...}
  T pop() {...}
  ...
}

String s = "Hello";
Stack<String> st =
  new Stack<String>();

st.push(s);

String t = st.pop();
```

Compilation: type erasure

```
class Stack<T> {
  void push(T o) {...}
  T pop() {...}
  ...
}

String s = "Hello";
Stack<String> st =
  new Stack<String>();

st.push(s);

String t = st.pop();
```

```
class Stack {
  void push(Object o) {...}
  Object pop() {...}
  ...
}

String s = "Hello";
Stack st = new Stack();

st.push(s);

String t = (String)st.pop();
```

Type Checking

```
interface Printable {
  void print();
}

class PrintableStack<T> implements Printable {
  void push(T o) {...}
  T pop() {...}

  void print() {
    ...
    for (T t : elems) {
      t.print();
    }
  }
}
```

```
PrintableStack<X> st = ..
st.push(x);
st.print();
```

Type Checking & Bounded Polymorphism

```
interface Printable {
  void print();
}
class PrintableStack<T implements Printable>
             implements Printable {
  void push(T o) {...}
  T pop() {...}

  void print() {
    ...
    for (T t : elems) {
      t.print();
    }
  }
}
```

```
PrintableStack<X> st = ..
st.push(x);
st.print();
```

Compilation

```
class Stack<T implements
             Printable> {
  void push(T o) {...}
  T pop() {...}

  void print() {
    ...
    for (T t : elems) {
      t.print();
    }
  }
}
```

```
class Stack {
  void push(Printable o) {..}
  Printable pop() {...}

  void print() {
    ...
    for (Printable t : elems)
      t.print();
    }
  }
}
```

Bounded Polymorphism

```
interface Comparable {
  int compareTo(Object other);
}
class Point implements Comparable {
  int compareTo(Object other) {
     Point otherAsPoint = (Point)other;
     return x == otherAsPoint.x && y == otherAsPoint.y;
  }
}
class PriorityQueue<T implements Comparable> {
  void add(T t) {
    T o = ...;
    ... t.compareTo(o) ...
  }
}
```

Bounded Polymorphism

```
interface Comparable<T> {
  int compareTo(T other);
}
class Point implements Comparable<Point> {
  int compareTo(Point other) {
    return x == other.x && y == other.y;
  }
}
class PriorityQueue<T implements Comparable<T>> {
  void add(T t) {
    T o = ...;
    ... t.compareTo(o) ...
  }
}
```

In Scala…

```
trait Comparable[T] {
  def compareTo(other : T): Boolean;
}
class Point extends Comparable[Point] {
  def compareTo(other : Point) = {
    x == other.x && y == other.y;
  }
}
class PriorityQueue[T <: Comparable[T]] {
  def add(t : T) = {
    val o : T = ...;
    ... t.compareTo(o) ...
  }
}
```

Wildcards

```
void printElements(Collection<Object> c) {
  for (Object e : c) {
    System.out.println(e);
  }
}

Collection<String> cs;
Collection<Integer> ci;
printElements(cs);
printElements(ci);
```

Wildcards

```
void printElements(Collection<?> c) {
  for (Object e : c) {
    System.out.println(e);
  }
}

Collection<String> cs;
Collection<Integer> ci;
printElements(cs);
printElements(ci);
```

Wildcards

```
void movePoints(Collection<? extends Point> c) {
  for (Point p : c) {
    p.move(10,10);
  }
}

Collection<Point> pts;
Collection<ColorPoint> cpts;
printElements(pts);
printElements(cpts);
```

Variance in Scala

```
Array[ColorPoint] <: Array[Point] ?

List[ColorPoint] <: List[Point]?

PartialFunction[Point,ColorPoint]
                <:
  PartialFunction[Point,Point] ?
```

Variance Annotations in Scala

- Class defined with covariant parameter T:

```
class List[+T] { ... }
```

- So `List[ColorPoint] <: List[Point]`
- Which of these is type-safe?

```
class List[+T] {
  def add(t : T) = ...
}
```

```
class List[+T] {
  def get() : T = ...
}
```

- Function Types:

```
class PartialFunction[-A,+R] { ... }
```

Variance Annotations in Scala

- Function Types:

```
class PartialFunction[-A,+R] { ... }
```

- Immutable Maps:

```
class Map[Key,+Value] extends PartialFunction[Key,Value]

Map[String,ColorPoint] <: Map[String,Point]
```

- Mutable Maps:

```
class Map[Key,Value] extends PartialFunction[Key,Value]

Map[String,ColorPoint] <: Map[String,Point]
```