SE580: Lecture 10

Overview

Java 1.5

Type erasure

Type inference

Formal model

Variance

CLR

Java 1.5

Generics in Java 1.5 is JSR14 from
http://www.jcp.org/jsr/detail/14.jsp

Features of Java 1.5:

Classes, interfaces and methods are generic, objects and fields are not.

No dependent types.

No use of `myType`.

F-bounded polymorphism.

Type inference for generic methods.

Implemented by *type erasure* (what is this?).

May or may not include *variance annotations* (what are these?).

Java 1.5

Draft specification Adding Generics to the Java Programmming Language by Bracha, Cohen, Kemper, Marx, Odersky, Panitz, Stoutamire, Thorup and Wadler.

Prototype implementation on Java Developer Network.

Type erasure (Section 6)

Replace a global type C<T_1,...,T_n> by what?

Replace a local type A by what?

```
interface Comparable<A> { int compareTo (A x); }
class SortedList<A extends Comparable<A>> {
  SortedList<A> insert (A x) { ... }
  A get (int i) { ... }
  int size () { ... }
  ...
}
...
SortedList<String> empty = new SortedList<String> ();
full = empty.insert ("hello").insert ("world");
for (int i=0; i<full.size (); i++) {
  String s = full.get (i);
  System.out.println (s);
}
```

Type erasure (Section 6)

What about inheritance?

```
class StringList extends SortedList<String> { }
...
// This may have been compiled with a Java 1.4 compiler!
// Worry about backward compatibility?
StringList empty = new StringList ();
full = empty.insert ("hello").insert ("world");
for (int i=0; i<full.size (); i++) {
  String s = full.get (i);
  System.out.println (s);
}
```

Type erasure (Section 6)

What about casting / instanceof / reflection?

```
SortedList<String> strings = new SortedList<String>.insert ("hello");
if (strings instanceof SortedList<Integer>) {
  SortedList<Integer> ints = (SortedList<Integer>)strings;
  Integer x = ints.get (0);
}
```

Type erasure (Section 6)

Even worse, what about arrays?

```
SortedList<Integer>[] intsArray = new SortedList<Integer>[1];
Object[] objectArray = intsArray;
SortedList<String> strings = new SortedList<String>.insert ("hello");
objectArray[0] = strings;
Integer x = intsArray[0].get (0);
```

Type inference (Section 5.6)

Original type inference algorithm:

1. Replace any missing type parameters with "*" (called Empty in Hobbes).

2. Make all types covariant with respect to *, that is $T[*] <: T[U]$ for any T and U.

3. Put in some restrictions that * can only be used once, to try to fix the bug (2) introduces.

```
static <A implements Comparable<A>> empty () { return new SortedList<A>() }
empty ().insert ("hello").insert ("world");
empty ().insert (new Integer (5));
empty ();
```

Type inference (Section 5.6)

3 didn't quite work...

```
static <A,B> A castit (B x) {
  final RWRef<A,B> r = build ();
  r.set (x);
  return r.get ();
}
static <A> Ref<A> build () { return new Ref<A> (); }
interface RWRef<A,B> {
  public A get ();
  public void set (B x);
}
class Ref<A> implements RWRef <A,A> {
    A contents;
    public void set (A x) { contents = x; }
    public A get () { return contents; }
}
```

Implementation has been fixed.

Spec is still incorrect.

Formal model

A significant part of Java 1.5 generics has been formally proven correct: Featherweight Java by Igarashi, Pierce and Wadler.

Variance

Introducing Variance into the Java programming Language by Hansen, von der Ahé Ernst, Torgersen and Bracha.

One possible solution to generics and arrays:

```
String[=] rwStrings = new String[1];
Object[=] rwObjects = new Object[1];
String[+] roStrings = rwStrings;  // OK!
Object[+] roStrings = roObjects;  // OK!
Object[-] woObjects = rwObjects;  // OK!
String[-] woStrings = woObjects;  // OK!
```

Which of these are OK?

```
rwStrings[0] = "hello";
rwObjects[0] = new Integer(5);
woStrings[0] = "hello";
woObjects[0] = new Integer(5);
roStrings[0] = "hello";
roObjects[0] = new Integer(5);
```

Variance

Do we need any run-time array assignment tests?

What happens if we try to replicate the problem example?

```
SortedList<=Integer>[=] intsArray = new SortedList<Integer>[1];
Object[=] objectArray = intsArray;
SortedList<String> strings = new SortedList<String>.insert ("hello");
objectArray[0] = strings;
Integer x = intsArray[0].get (0);
```

Hooray, generic arrays!

Variance

You can do this to other generic classes too, not just arrays:

```
SortedList<=Point> rwPoints = new SortedList<Point> ();
SortedList<+Point> roPoints = rwPoints;
SortedList<-ColoredPoint> woCPoints = rwPoints;
woCPoints.insert (new ColoredPoint (5,37,red));
Point p = roPoints.get (0);
```

Deemed ‘too complex’ by Sun...

CLR

Generics in .NET at http://research.microsoft.com/projects/clrgen/.

Implementation paper Design and Implementation of Generics for the .NET CLR by Kennedy and Syme.

Classes, interfaces and methods are generic, objects and fields are not.

No dependent types.

No use of myType.

F-bounded polymorphism.

Main difference: no type erasure!

CLR

Types are instantiated at run-time to produce new Class objects.

What happens if we try to use instanceof and casting?

```
SortedList<String> strings = new SortedList<String>.insert ("hello");
if (strings instanceof SortedList<Integer>) {
  SortedList<Integer> ints = (SortedList<Integer>)strings;
  Integer x = ints.get (0);
}
```

Opportunities for optimizing generic code at run-time.

Uses a flyweight pattern to avoid code bloat.

Summary

Advanced generics systems are coming out of the research labs.

Java 1.5 and CLR 2.0 have similar generics, the main difference is type erasure.

Next week: Final exam.