SE580: Lecture 9

Overview

The binary method problem

A solution: myType

Generic Java

Type inference for generics

The binary method problem

Consider:

```
class Point {
  field x : Integer; field y : Integer;
  method infix <= (p : Point) : Boolean {
    return this.x < p.x || (this.x == p.x && this.y <= p.y);
  }
}
class ColoredPoint extends Point {
  field color : Color;
  method infix <= (p : ColoredPoint) : Boolean {
   return this.x < p.x || (this.x == p.x && ( this.y < p.y || this.y == p.y && this.color <= p.color ) );
  }
}
```

This code does not compile. Why not?

This is an instance of the *binary method problem*.

The binary method problem

Java's solution (almost):

```
interface Comparable { method infix <= (p : Comparable) : Boolean; }
class Point implements Comparable {
  field x : Integer; field y : Integer;
  method infix <= (p : Comparable) : Boolean {
    return this.x < p.x || (this.x == p.x && this.y <= p.y);
  }
}
class ColoredPoint extends Point {
  method infix <= (p : Comparable) : Boolean {
   return this.x < p.x || (this.x == p.x && ( this.y < p.y || this.y == p.y && this.color <= p.color ) );
  }
}
```

This code does not compile. Why not? What can we do about it?

Introducing myType

Another possible solution:

```
class Point {
  field x : Integer; field y : Integer;
  method infix <= (p : myType) : Boolean {
    return this.x < p.x || (this.x == p.x && this.y <= p.y);
  }
}
class ColoredPoint extends Point {
  field color : Color;
  method infix <= (p : myType) : Boolean {
   return this.x < p.x || (this.x == p.x && ( this.y < p.y || this.y == p.y && this.color <= p.color ) );
  }
}
```

What is myType doing here?

Introducing myType

Another example: cloning. How does Java handle cloning?

An alternative:

```
class Point {
  ...
  field cloner : Cloner[myType];
  method clone () : myType { return cloner.clone (this); }
}
interface Cloner [type a] { method clone (x : a) : a; }
class PointFactory implements Cloner[Point] {
  method build (x : Integer, y : Integer) : Point { ... }
  method clone (p : Point) : Point { ... }
}
```

How can we finish this example?

If p:Point we can now write let q : Point = p.clone ();

A problem with myType

```
class Foo {
  method m (x : myType) : Integer { return 0; }
}
class Bar extends Foo {
  field b : Integer;
  method m (x : myType) : Integer { return x.b; }
}
thread Main {
  let x : Foo = new Foo {};
  let y : Bar = new Bar { b = 37 };
  let z : Foo = y;
  z.m (x);
}
```

What is the problem? How can we fix this?

Semantics of myType

Classes now declare a myType type variable:

```
class Foo extended myType {
  // myType <# Foo is a local type variable here.
}
```

What changes to the syntax / dynamic semantics / static semantics are needed?

Homework

Last homework! Implement `myType`.

Another solution: F-bounded polymorphism

Another solution (adopted by Generic Java):

```
interface Comparable [type a] { method infix <= (x : a) : Boolean; }
class Foo implements Comparable[Foo] { ... }
class Bar implements Comparable[Bar] { ... }
interface SortedList [type a implements Comparable[a]] extends List[a] {
  method insert (x : a) : SortedList[a];
}
```

How can we implement SortedList?

What about cloning?

Type inference for generics

An example Java 1.5 program:

```
public class Example {
    public static void main (String [] args) {
        Ref<String> r = build ();
        r.set ("hello");
        System.out.println (r.get ());
    }
    static <A> Ref<A> build () {
        return new Ref<A> ();
    }
    static <A> Ref<A> build (A x) {
        Ref<A> result = build (); result.set (x); return result;
    }
}
class Ref<A> {
    A contents;
    public void set (A x) { contents = x; }
    public A get () { return contents; }
}
```

Note the code: `build ()` what is odd here?

Type inference for generics

Generic Java does *type inference* on generic methods.

You write:

```
Ref<String> a = build ();
Ref<String> b = build ("hello");
```

The compiler figures out:

```
Ref<String> a = build<String> ();
Ref<String> b = build<String> ("hello");
```

Why does Java include type inference for generic methods?

Type inference for generics

Two strategies for type inference for generic methods:

1. C++ bottom-up.

2. Standard ML constraint-solving (the *Hindley-Milner algorithm*).

What are the tradeoffs here?

Next week

Concurrency.