SE580: Lecture 8

Overview

Generics

Syntax

Dynamic semantics

Static semantics

Implementation

Homework

Generics

What are generics? Why are they interesting?

Two possible implementations of generics: *templates* and *parametric polymorphism*.

What are these? What are the tradeoffs between them?

Does C++ support generics? Java? C#?

Generics

An example generic class in Hobbes:

```
interface Ref[type a] {
  method get () : a;
  method set (n : a);
}
class RefImpl[type b] implements Ref[b] {
  mutable field contents : b;
  method get () : b { return this.contents; }
  method set (n : b) { this.contents := n; }
}
```

What does this do? What is the scope of type a? Type b?

How could we code generic lists?

Generics

Part of the design space for generics: *what things can be generic* and *what things can you be generic in*?

What are the answers for C++? For Generic Java?

Generics

Example: generic *objects*:

```
interface List[type a] { ... }
class EmptyList[type a] implements List[a] { ... }
object Empty[type a] : EmptyList[a] {}
```

The use of templates like this is called *value polymorphism*.

Does C++ allow value polymorphism? Does Generic Java?

What is good about value polymorphism? What is bad?

Generics

Example: templates parameterized on *data values*:

```
interface Array[type a, field size : Integer] {
  method set (index : int, value : a);
  method get (index : int) : a;
  ...
}
```

Templates like this are called *dependent types*.

Does C++ allow dependent types? Does Generic Java?

What is good about dependent types? What is bad?

Generics

Example: *nested* templates:

```
interface Converter[type a, type b] {
  method a2b (x : a) : b;
  method b2a (y : b) : a;
}
interface ConvertableRef[type a] extends Ref[a] {
  method convert [type b] (c : Converter[a,b]) : ConvertableRef[b];
}
```

Nested templates like this are called *impredicative*.

Is C++ impredicative? Is Generic Java?

What is good about impredicative types? What is bad?

Generics

We now extend Hobbes...

Interfaces, classes and objects can be generic (not methods!).

They are generic in types (not objects!).

Jargon: Hobbes supports non-dependent, predicative, parametric value polymorphism.

Syntax of generics

We now have *local* as well as *global* types.

Global type and object definitions now have *type parameters*.

Global type uses now have *type arguments*.

What changes to the syntax do we need to make?

Dynamic semantics of generics

Consider this program:

```
class Ref[type a] {
  mutable field contents : a;
  method get () : a { return this.contents; }
  method set (n : a) { this.contents := n; }
}
object R : Ref[Integer] { contents = 37 }
thread Main { R.set (5); }
```

How does this execute?

What changes to the dynamic semantics do we need to make to support this?

Static semantics of generics

Consider this program:

```
class Ref[type a] {
  mutable field contents : a;
  method get () : a { return this.contents; }
  method set (n : a) { this.contents := n; }
}
object R : Ref[Integer] { contents = 37 }
thread Main { R.set (5); }
```

Why does this type check?

What changes to the static semantics do we need to make to support this?

Static semantics of generics

```
class RefImpl [type a] {
  mutable field contents : a;
}
interface Maybe [type a] {
  method get () : a;
}
class MaybeYes [type a] implements Maybe[a] {
  field contents : a;
  method get () : a { return this.contents; }
}
class MaybeNo [type a] implements Maybe[a] {
  method get () : a { return Error! "No such element."; }
}
object Null [type a] : MaybeNo [a] {}
object R : RefImpl[Maybe[Integer]] { contents=Null[Integer] }
thread Main {
  // Hooray, we have a null pointer! :-)
  R.contents = new MaybeYes[Integer] { contents = 37 };
  let x : Integer = R.contents.get ();
}
```

Static semantics of generics

```
class RefImpl [type a] {
  mutable field contents : a;
}
interface Maybe [type a] {
  method get () : a;
}
class MaybeYes [type a] implements Maybe[a] {
  field contents : a;
  method get () : a { return this.contents; }
}
class MaybeNo [type a] implements Maybe[a] {
  method get () : a { return Error! "No such element."; }
}
object Null [type a] : MaybeNo [a] {}
object R[type a] : RefImpl[maybe[a]] { contents=Null[a] }
thread Main {
  // Oh dear :-(
  R[Integer].contents = new MaybeYes[Integer] { contents = 37 };
  let x : String = R[String].contents.get ();
}
```

Implementation

What changes need made to the interpreter?

Homework

As per usual...

Next week

The binary method problem.

Type inference and Generic Java