SE580: Lecture 7

Overview

Subclassing and subtyping

Syntax

Dynamic semantics

Static semantics

Subtyping

Subclassing

Implementation

Homework

Subclassing and subtyping

What is subclassing (aka inheritance)? Why is subclassing useful?
What support for subclassing is provided by Java?

What is subtyping (aka polymorphism)? Why is subtyping useful?
What support for subtyping is provided by Java?

What about Hobbes?

Syntax for subclassing and subtyping

We need extra syntax for subclassing and subtyping, for example:

```
interface Foo { method a () : Integer; }
interface Bar extends Foo { method b () : Integer; }
class FooImpl implements Foo { method a () : Integer { return 4; } }
class BarImpl extends FooImpl implements Bar { method b () : Integer { return 5; } }
```

What do we need to add?

Dynamic semantics of subclassing

We need an extra dynamic rule to cope with subclassing, for example:

```
  interface Foo { method a () : Integer; }
  interface Bar extends Foo { method b () : Integer; }
  class FooImpl implements Foo { method a () : Integer { return 4; } }
  class BarImpl extends FooImpl implements Bar { method b () : Integer { return 5; } }
  object B : BarImpl { }
  thread Main { let x = B.a (); }
-->
  ???
```

What do we need to add?

Static semantics of subclassing and subtyping

Much trickier!

Some issues (for each issue: What is it? Why does it matter? What does Java do?):

1) Multiple inheritance.

2) Should subclassing imply subtyping?

3) Is subtyping *structural* or *named*?

4) Method overriding.

5) Field overriding.

Static semantics of subclassing and subtyping

Hobbes has both *subtypes* $T <: U$ and *subclasses* $T <\# U$.

We use judgements:

Subclassing: $\Gamma \vdash T <\# U$

Subtyping: $\Gamma \vdash T <: U$

Field overriding: $\Gamma \vdash F <: G$

Method overriding: $\Gamma \vdash M <: N$

What are these judgements for?

Subtyping should satisfy *subsumption*: if $T <: U$ then it is safe to treat any value of type T as if it had type U.

Subtyping

```
interface Foo {
  method foo ();
}
interface Bar extends Foo {
  method foo ();
  method baz ();
}
```

Is Bar <: Foo?

Subtyping

```
interface Foo {
  method foo ();
}
interface Bar {
  method foo ();
  method baz ();
}
```

Is Bar $<:$ Foo?

Subtyping

```
interface Foo {
  method foo ();
  method bar ();
}
interface Bar extends Foo {
  method foo ();
  method baz ();
}
```

Is Bar <: Foo?

Subtyping

```
interface Foo {
  method foo () : String;
}
interface Bar extends Foo {
  method foo () : Integer;
}
```

Is Bar <: Foo?

Subtyping

```
interface Sup {}
interface Sub extends Sup {}
interface Foo {
  method foo () : Sub;
}
interface Bar extends Foo {
  method foo () : Sup;
}
```

Is Bar <: Foo?

Subtyping

```
interface Sup {}
interface Sub extends Sup {}
interface Foo {
  method foo () : Sup;
}
interface Bar extends Foo {
  method foo () : Sub;
}
```

Is Bar $<:$ Foo?

What does Java do? What is the 'right' thing to do? Does this come up in practice?

Subtyping

```
interface Foo {
  method foo (x : Integer);
}
interface Bar extends Foo {
  method foo (x : String);
}
```

Is Bar <: Foo?

Subtyping

```
interface Sup {}
interface Sub extends Sup {}
interface Foo {
  method foo (x : Sup);
}
interface Bar extends Foo {
  method foo (x : Sub);
}
```

Is Bar <: Foo?

Subtyping

```
interface Sup {}
interface Sub extends Sup {}
interface Foo {
  method foo (x : Sub);
}
interface Bar extends Foo {
  method foo (x : Sup);
}
```

Is Bar $<:$ Foo?

What does Java do? What is the 'right' thing to do? Does this come up in practice?

Subtyping

Jargon alert!

Given a type $T[U]$ which contains a type U.

$T[U]$ is *covariant* in U if $U_1 <: U_2$ implies $T[U_1] <: T[U_2]$.

$T[U]$ is *contravariant* in U if $U_1 <: U_2$ implies $T[U_1] :> T[U_2]$.

$T[U]$ is *invariant* in U otherwise.

Subtyping

Examples:

1) method m () : U; // is U covariant, contravariant or invariant?
2) method m (x : U); // is U covariant, contravariant or invariant?
3) method m (x : U) : U; // is U covariant, contravariant or invariant?

What should the general rule look like for methods?

If ...
then $\Gamma \vdash$ method $x.m\ (x_1{:}T_1,...,x_n{:}T_n) : T$
$<:$ method $x.m\ (x_1{:}U_1,...,x_n{:}U_n) : U$.

What should the general rule look like for interfaces?

If $\Gamma \vdash T_1$: interface ... $\Gamma \vdash T_n$: interface
and $\Gamma \vdash M_1$: method in c ... $\Gamma \vdash M_i$: method in c
and ...
then $\Gamma \vdash$ interface c extends $T_1,...,T_n\ \{\ M_1,...,M_i\ \}$: declaration.

How do we define $\Gamma \vdash T <: U$?

Subtyping

Problem example from Java:

```
U[ ] // is U covariant, contravariant or invariant?
```

Example: if Triangle <: Shape then is Triangle[ ] <: Shape[ ]?

Oh dear...

```
Triangle[ ] triangles = new Triangle[1];
Shape[ ] shapes = triangles;
Square square = ...;
shapes[0] = square;
Triangle triangle = triangles[0];
```

Subtyping

What changes do we need to make to the static semantics to typecheck code like this:

```
method foo (list : ShapeList, triangle : Triangle, square : Square) : ShapeList {
  let tmp1 : ShapeList = list.cons (triangle);
  let tmp2 : ShapeList = tmp1.cons (square);
  return tmp2;
}
```

where:

```
interface Shape { ... }
interface Triangle extends Shape { ... }
interface Square extends Shape { ... }
interface ShapeList { ... method cons (x : Shape) : ShapeList; ... }
```

Subclassing and subtyping

```
interface Foo {
  method foo ();
}
class Bar implements Foo {
}
```

Is Bar $<:$ Foo?

Subclassing and subtyping

```
interface Foo {
  method foo ();
}
class Bar implements Foo {
  method foo () {}
}
```

Is Bar <: Foo?

Subclassing and subtyping

```
class Foo {
}
class Bar extends Foo {
}
```

Is Bar <# Foo?

Is Bar <: Foo?

Subclassing

```
class Foo {
  field foo : Integer;
}
class Bar extends Foo {
  field foo : String;
}
```

Is Bar <# Foo?

Subclassing

```
interface Sup {}
interface Sub extends Sup {}
class Foo {
  field foo : Sup;
}
class Bar extends Foo {
  field foo : Sub;
}
```

Is Bar <# Foo?

What does Java do? What is the ‘right’ thing to do? Does this come up in practice?

Subclassing

```
interface Sup {}
interface Sub extends Sup {}
class Foo {
  field foo : Sub;
}
class Bar extends Foo {
  field foo : Sup;
}
```

Is Bar <# Foo?

Subclassing

```
interface Sup {}
interface Sub extends Sup {}
class Foo {
  mutable field foo : Sup;
}
class Bar extends Foo {
  mutable field foo : Sub;
}
```

Is Bar <# Foo?

Subclassing

For fields...

1) immutable field f : U; // Is U covariant, contravariant or invariant?
2) mutable field f : U; // Is U covariant, contravariant or invariant?

What should the general rules look like for immutable fields?

If ...
then $\Gamma \vdash$ immutable field f : T $<:$ immutable field f : U.

What about mutable fields?

If ...
then $\Gamma \vdash$ mutable field f : T $<:$ mutable field f : U.

Subclassing

What should the general rule look like for classes?

> If $\Gamma \vdash T_0$: class, $\Gamma \vdash T_1$: interface ... $\Gamma \vdash T_n$: interface
> and $\Gamma \vdash M_1$: method in c ... $\Gamma \vdash M_i$: method in c
> and $\Gamma \vdash F_1$: field ... $\Gamma \vdash F_j$: field
> and ...
> then $\Gamma \vdash$ class c extends T_0 implements T_1,...,T_n { F_1,...,F_j M_1,...,M_i } : declaration.

How do we define $\Gamma \vdash T <\# U$?

Implementation

Routine changes to the parser and interpreter.

What needs added to the typechecker?

Homework

Complete the typechecker for Hobbes (currently missing most of the cases for subtyping and subclassing).

Next week

Generics.