CSC547: Type Systems for OO Languages

Advanced class-based features

Abadi and Cardelli, Chapter 3

Object types

Type parameters

The binary method problem
Object types

In Java, object types are called interfaces:

```java
interface Cell {
    int contents;
    void set (int n);
    int get ();
}
class cell implements Cell {
    int contents = 0;
    void set (int n) { this.contents = n; }
    int get () { return this.contents; }
}
```

(Well almost...)

Imagine if Java had a rule that interfaces were types, but classes weren’t, for example:

```java
Cell okCell = new cell (); // allowed
cell badCell = new cell (); // not allowed
```

What would be the tradeoffs?
Object types

Consider the pseudo-Java:

interface ReCell extends Cell {
    int backup;
    void restore ();
}
class reCell extends cell implements ReCell {
    int backup = 0;
    void set (int n) { ... }
    void restore () { ... }
}

Cell foo = new cell ();
ReCell bar = new reCell ();
foo = bar;

This separates implementation issues (classes) from typing issues (interfaces).

Is this good or bad?
Object types

From a class definition, it is easy to read off the corresponding interface:

class Point {
    int x=0; int y=0;
    int getX () { return this.x; }
    int getY () { return this.y; }
    void move (int dX, int dY) { this.x += dX; this.y += dy; }
}

What is ObjectTypeOf (Point)?
Distinguishing subclassing from subtyping

An easy definition of subtyping on object types:

\[ O' <: O \] whenever \( O' \) has the same components as \( O \) and possibly more

This rule is called *shallow structural subtyping*.

(This isn’t the rule Java uses, but it will do for now...)

The important property of subtyping compared to subclassing:

If \( A \) is a subclass of \( B \) then

\[ \text{ObjectTypeOf}(A) <: \text{ObjectTypeOf}(B) \]

This is ‘subclassing-implies-subtyping’.

How is this different from ‘subclassing-is-subtyping’?
Distinguishing subclassing from subtyping

Consider:

```java
class a {
    int foo (int x, int y) { return x+y; }
    int baz (int x) { return 2*x; }
}
class b {
    int foo (int x, int y) { return x*y; }
}
class c extends b {
    int baz (int x) { return -x; }
}
```

What are the object types for these classes?

How are the object types related by subtyping?

How are the classes related by subclassing?
Type parameters

C++-like languages support templates, aka generic classes, aka parameterized classes, aka polymorphic classes, aka...

```java
interface Array <Element> {
    Element get (int i);
    void set (int i, Element e);
}
```

How could we implement the Array interface?

```java
class array <Element> implements Array <Element> {
    ...
}
```

Why are type parameters useful?
Type Parameters

Consider (where Vegetables <: Food):

```java
interface Person {
...
  void eat (Food stuff);
}

interface Vegetarian extends Person {
...
  void eat (Vegetables stuff);
}
```

Will this succeed?

How can type parameters help?
Type parameters

The *binary method problem* comes up whenever we have an interface with a method which operates on objects of the same class:

```java
interface Foo {
    void bar (Foo f);
}
```

for example:

```java
interface Max {
    Max max (Max other);
}
```

class SortedTree <Element <: Max> {
    ...
}

class maxClass implements Max {
    int n = 0;
    Max (Max other) {
        ...what code should go here?...
    }
}
```

Can we do this without casting?
Subclassing without subtyping

Some ‘natural’ language extensions to Java break subclassing-is-subtyping.

Imagine we had a This type in Java meaning ‘the type of the current class’.

```java
interface MaxInt {
    int n;
    Max max (Max other);
}

class maxClass implements MaxInt {
    int n = 0;
    This max (This other) {
        ...what code should go here?...
    }
}
```

Seems like it might be useful...
Subclassing without subtyping

```java
interface MinMaxInt {
    int n;
    MinMax min (MinMax other);
    MinMax max (MinMax other);
}

class minMaxClass extends maxClass implements MinMaxInt {
    This min (This other) {
        ...what code should go here?...
    }
}

Er but do we have MinMaxInt <: MaxInt?

Ooops, adding This breaks ‘subclassing-implies-subtyping’.
Operator protocols

One possibility:

    interface MaxProtocol <X> {
        X max (X other);
    }

then rather than:

    class SortedTree <Element <: Max> { ... }

we have:

    class SortedTree <Element <: MaxProtocol <Element>> { ... }

and then:

    class maxClass implements MaxProtocol <maxClass> {
        ... what goes here? ...
    }

This is called \textit{F-bounded polymorphism}, and is one of the proposals for inclusion in Java 3.
Operator protocols

Then we have:

```java
interface MinMaxProtocol <Element> extends MaxProtocol <Element> {  
    Element max (Element other);
}

class minMaxClass extends maxClass implements MinMaxProtocol <minMaxClass> {  
    ... what goes here? ...
}

Hooray, subclassing, subtyping etc. all works!
```
Operator protocols

Alternatively, use *subprotocols*.

Rather than:

```java
class SortedTree <Element <: MaxProtocol <Element>> { ... }
```

we could have:

```java
class SortedTree <P <: MaxProtocol> { ... }
```

and instantiate `SortedTree <MinMaxProtocol>`. This is called *higher-order type parameters* and is not usually seen outside the research lab...