CSC547: Type Systems for OO Languages

Class-based languages

Abadi and Cardelli, Chapter 2

Classes, objects method lookup.

Subclassing, subtyping.

Covariance, contravariance.

Specialization.

(We will use pseudo-Java syntax rather than pseudo-Modula 3.)
Classes, objects and method lookup

Consider the pseudo-Java:

class cell {
    int contents = 0;
    int get () { return this.contents; }
    void set (int n) { this.contents = n; }
}

    cell foo = new cell ();
    cell bar = new cell ();

How is memory allocated for foo and bar?

*Hint:* there’s two ways to do it: embedded methods and delegated methods.

What are the tradeoffs between embedded and delegated methods?

*Hint:* space-time tradeoff.

Can the programmer distinguish between embedded and delegated methods?

*Hint:* method update.
Subclasses and inheritance

Consider the pseudo-Java:

```java
class reCell extends cell {
    int backup = 0;
    void set (int n) {
        this.backup = this.contents;
        super.set (n);
    }
    void restore () {
        this.contents = this.backup;
    }
}

cell foo = new cell ();
reCell bar = new reCell ();
```

How is memory allocated now for foo and bar?

*Hint*: there’s three ways to do it: *embedded methods*, *hierarchical method suites* and *collapsed method suites*.

What are the tradeoffs between these approaches?
Multiple inheritance

Consider the following pseudo-Java:

class A {
    String bad () { return "A"; }
}
class B extends A {
    String bad () { return "B"; }
}
class C extends A {
    String bad () { return "C"; }
}
class D extends B, C {
}

D foo = new D ();
print (foo.bad ());

What happens here?

Should this program be allowed?
Dynamic dispatch

In pseudo-Java:

cell myCell = new cell ();
reCell myReCell = new reCell ();
myCell.set (3);
myCell = myReCell;
myCell.set (3);

What happens here?

*Hint*: dynamic method dispatch (CSC447).

In an imperative language, would the assignment `myCell = myReCell` be allowed?

In OO languages `reCell` is a *subtype* of `cell`. 
Subtyping vs subclassing

In Java-like languages, subtyping and subclassing are the same thing:

\[
\text{Foo is a subtype of Bar if and only if } \\
\text{Foo extends Bar}
\]

but this means we’re combining lots of features:

- Inheritance and overriding
- Subtyping

‘Classic’ OO languages have subtyping == subclassing.

‘Modern’ OO languages have subtyping != subclassing.

What are the tradeoffs?
Subsumption

Write $A <: B$ for ‘$A$ is a subtype of $B$’.

Subsumption says:

If $a : A$ and $A <: B$ then $a : B$

For ‘classic’ class-based languages:

Class $A$ is a subclass of $B$ if and only if $A <: B$

This is called ‘subclassing-is-subtyping’

Consider:

```java
cell foo = new reCell();
reCell bar = new cell();
```

Do these typecheck? Why?
Type information, lost and found

Consider the pseudo-Java:

```java
void foo (cell x) {
    if (x.instanceOf (reCell)) {
        ((reCell)x).restore ();
    } else {
        x.set (3);
    }
}
```

What does this code do?

Is `instanceOf` (normally called `typecase`) a good idea?

How could we get by without `instanceOf`?
Covariance, contravariance and invariance

In ML (and other similar languages) there is a type $A \times B$ of pairs with:

- $(a,b) : A \times B$ whenever $a : A$ and $b : B$
- $\text{lft}(a) : A$ whenever $a : A \times B$
- $\text{snd}(a) : B$ whenever $a : A \times B$

How would we implement a pair class in Java? Or in C++?

Which of these should typecheck?

```java
String*String foo = ("hello", "world");
Object*Object bar = foo;
String*String baz = bar;
```

If $A <: A'$ and $B <: B'$ then what is the relation between $A \times B$ and $A' \times B'$?
Covariance, contravariance and invariance

In ML (and other similar languages) there is a type A ref of references with:

- \( \text{ref}(a) : \text{A ref} \) whenever \( a : \text{A} \)
- \( !a : \text{A} \) whenever \( a : \text{A ref} \)
- \( a := b : \text{A} \) whenever \( a : \text{A ref} \) and \( b : \text{A} \)

How would we implement a reference class in Java? Or in C++?

Does the following typecheck?

If it does typecheck, what does it do?

```
String ref fred = ref("hello");
Object ref wilma = fred;
wilma := new Integer(37);
String betty = !fred;
```

If \( A <: A' \) and \( B <: B' \) then what is the relation between \( \text{A ref} \) and \( \text{A'} \text{ ref} \)?
Covariance, contravariance and invariance

In ML (and other similar languages) there is a type A -> B of functions with:

- f : A->B whenever f is declared B f (A a) { ... }
- f(a) : B whenever f : A->B and a : A

How would we implement a function class in Java? Or in C++?

Which of these should typecheck (assuming Integer <: Float < Object?)

Float double (Float x) { return 2.0*x; }
Float->Float foo = double;
Float->Object bar = foo;
Float->Integer baz = foo;

Object->Float bar = foo;
Integer->Float baz = foo;

If B <: B' then what is the relation between A->B and A->B'?

If A <: A' then what is the relation between A->B and A'->B?
Covariance, contravariance and invariance

Consider the pseudo-Java:

```java
interface Foo <A,B,C> {
    A bar (B x);
    C baz (C y);
}
```

Is A covariant, contravariant or invariant?
Is B covariant, contravariant or invariant?
Is C covariant, contravariant or invariant?
What is the general rule for co/contra/invariance?
Method specialization

Consider the pseudo-Java:

```java
class A { ... }
class B extends A { ... }
class C extends B { ... }

class Foo { B fish (B x) { ... } }
```

Which of the following are typesafe?

```java
class Bar extends Foo { A fish (B x) { ... } }
class Bar extends Foo { B fish (B x) { ... } }
class Bar extends Foo { C fish (B x) { ... } }
class Bar extends Foo { B fish (A x) { ... } }
class Bar extends Foo { B fish (B x) { ... } }
class Bar extends Foo { B fish (C x) { ... } }
```

What is the general rule for method specialization?
Self specialization

Consider the pseudo-Java:

class cell {
    int contents = 0;
    int get () { return this.contents; }
    void set (int n) { this.contents = n; }
}

class reCell extends cell {
    int backup = 0;
    void set (int n) { this.backup = this.contents; super.set (n); }
    void restore () { this.contents = this.backup; }
}

In cell’s get method, what is the type of this?

In reCell’s get method, what is the type of this?

Is this being specialized covariantly or contravariantly?