Lecture 36: Java 5 types

CSC 131
Fall, 2008

Kim Bruce

Constraining Genericity

Need to constain type parameters:

```
interface Comparable {
    boolean equal(Comparable other);
    boolean lessThan(Comparable other);
}

class BST<T extends Comparable> { ... }

class OrderedRecord implements Comparable {
   ... // inst vble declarations
   boolean lessThan(Comparable other) {
      ???
   }
}
```

F-Bounded Quantification

- Mitchell et al introduced F-bounded quantification

```
interface Comparable<T> {
    boolean equal(T other);
    boolean lessThan(T other);
}

class BST<T extends Comparable<T>> { ... }

class OrderedRecord
             implements Comparable< OrderedRecord> {
   boolean lessThan(OrderedRecord other) {
      if (...)...
   }
}
```

F-Bounded Quantification

- Seems to solve the problem, but sometimes too complex to write easily.

```
public class ComparableAssoc
     <Key extends Comparable<Key>, Value>
implements Comparable<ComparableAssoc<Key,Value>> {
```

- Not preserved by subclasses.
  - Suppose C extends Comparable<C> and D extends C
  - Then D extends Comparable<C> but not Comparable<D>
- See Bruce, "Some Challenging Typing Issues in Object-Oriented Languages" on my web pages under recent papers.

Also Polymorphic Methods

```
interface Visitor<T> {
    T visitNumber(int n);
    T visitSum(T left, T right);
}

class Expr {
    public <T> T accept(Visitor<T> v);
}

class Number extends Expr {
    private int n;
    public Number(int n) { this.n = n; }
    public <T> T accept(Visitor<T> v) {
      return v.visitNumber(this.n); }
}
```

Java Wild Cards

- Four ways to specify type parameters :

  T : fixed type
  ? extends T : some extension of T,
  ? super T : some type that T extends,
  ? : any type

- Examples:

  C<? extends T>: *can be C<U> for any U extending T.*

  C<? super T>: *can be C<U> for any U s.t. T extends U.*

  C<?>: *can be C<U> for any U.*

Example

- In class TreeSet<E>:
  - boolean addAll(Collection<? extends E> c)
  - *constructor*: TreeSet(Comparator<? super E> c)
  - Comparator <? super E> comparator( )
  - *where* interface Comparator<T> has method
    int compare(T o1, T o2)

In libraries almost all occurrences are of form ? extends E *or just* ?, *and are in parameter position.*

What do wildcards mean?

`C<? extends T>` ≡ ∃(t<:T). C<t>

`C<? super T>` ≡ ∃(t:>T). C<t>

`C<?>` ≡ ∃t. C<t>

Compare with

`C<t extends T>` ≡ ∀(t<:T). C<t>

Wildcard Restricts Usage

- If `ds : List<? extends T>`
  ≡ ∃t *extends T.List*<t>
  then can access elements, but not insert.
- More carefully, if `List<T>` has methods

  `get: () → T, set: T → void`

  then

  `ds.get()` will return value of type `T`, but
  `ds.set(o)` *always illegal*, no matter what type of `o`.
  I.e., ds is *read-only*

Restrictions Confusing

- ?s are not equal to each other or even itself:

```
public void twiddle(Stack<?> s) {
   if (!s.empty())
      s.push(s.pop());
}
```

- Illegal, because type of `s.pop()` not recognized as same as argument type of `s.push(...)`.
- Can't even write swap!
- Can fix by calling polymorphic method where type given a name.

Avoiding Wildcards

- Recall from logic, if B does not contain t then

  ∀t.(A(t) → B) ≡ (∃t.A(t))→ B

- Thus by "Curry-Howard equivalence"

```
<T extends C> void m(List<T> aList){...}
```

is equivalent to

```
void m(List<? extends C> aList){...}
```

- However, there is no equivalent for return type or types of fields.

Are Wild-Cards Worth It?

- They show up in all of the Collection classes:

```
public ArrayList( Collection<? extends E> c )
public void addAll( Collection<? extends E> c )
public void removeAll( Collection<?> c )
```

- Can be replaced by similar:

```
public ArrayList<T extends E>( Collection<T> c )
public <T extends E> void addAll( Collection<T> c )
public <T> void removeAll( Collection<T> c )
```

- Provides more information: *Can write swap!*