CS2030S Recitation Problem Set 4

Brian Cheong

Recap

Wildcards

- A substitute for any type
- Can be upper bounded (`? extends T`)
  - ? can be T and it's subtypes
  - This gives rise to covariance behaviour
- Can be lower bounded (`? super T`)
  - ? can be T and it's supertypes
  - This gives rise to contravariance behaviour
- Use unbounded when you know nothing. Better than rawtyping

PECS

- Think from the perspective of the container
- Producer Extend
  - If you say you produce T, makes sense that you produce subtypes of T
- Consumer Super
  - You have a T, makes sense to store T in containers of T and containers of supertype of T

Type inference

- Very algorithmic (easy to score)
- Find constraints (each one produces a set of types)
  - Argument Type: Is there wildcards used for the T in the argument?
  - Target Type: Is T going to be bound to some type? Integer i = myObj<Integer>.foo()
  - Bounds on Type Parameters: Does T extend/super something?
- Solve the constraints
  - Ignore subclasses not specified in constraints
  - Solution may be a superclass of the types in constraints

Q1a

```
class B<T> {
  T x;
  static T y;
}
```

Run this and explain

- Throws compile error
- T is instantiated when an object B is created
- Static means no object is created, so T is not instantiated

Q1b

```java
class C<T> {
  static int b = 0;
  C() {
    this.b++;
  }
  public static void main(String[] args) {
    C<Integer> x = new C<>();
    C<String> y = new C<>();
    System.out.println(x.b);
    System.out.println(y.b);
  }
}
```

- prints 2 on both lines
- Remember that there is still only one class C
- `int b` is for that class C

Q2

- Determine subtyping
  - Typing is just a relation (partial-order)
- Draw $S \to T$, if $S <: T$, Hasse diagram
- Can omit the transitive subtyping
- *Brian will now demonstrate his artistic skills*

Q3

```
static <T extends Comparable<T>> T max(List<T> list) {
  T max = list.get(0);
  if (list.get(1).compareTo(max) > 0) {
    return list.get(1);
  }
  return max;
}

class Fruit implements Comparable<Fruit> {
  public int compareTo(Fruit f) {
    return 0; // stub
  }
}

class Apple extends Fruit {
}
```

Q3a

What would T be inferred as if we call `Fruit f = max(fruits)`

- Target Typing: `T <: Fruit`
- Argument Typing: `List<Fruit> <: List<T>` $\Longrightarrow$ `T = Fruit`
- Bounds on T: `T <: Comparable<T>`

Q3bi

Why does it fail to compile if we call `Fruit f = max(apples)`

- Target Typing: `T <: Fruit`
- Argument Typing: `List<Apple> <: List<T>` $\Longrightarrow$ `T = Apple`
- Bounds on T: `T <: Comparable<T>`
- If T = Apple, does `Apple <: Comparable<Apple>` hold?
- No, that's why we die. Apple is a Comparable<Fruit>

Q3bii

Why does it fail to compile if we call `Apple a = max(Apples)`

- Target Typing: `T <: Apple`
- Argument Typing: `List<Apple> <: List<T>` $\Longrightarrow$ `T = Apple`
- Bounds on T: `T <: Comparable<T>`
- Same as before, we die because `Apple </: Comparable<Apple>`

Q3biii

Why does it fail to compile if we call `Apple a = max(Fruits)`

- Target Typing: `T <: Apple`
- Argument Typing: `List<Fruit> <: List<T>` $\Longrightarrow$ `T = Fruit`
- Bounds on T: `T <: Comparable<T>`
- Why do we die?
- `Fruit </: Apple`

Q3c

- How do we fix this? (Just change the method header)
- Remember the main issue is that `Apple </: Comparable<Apple>`
- If only there was something could put to make it more flexible, I would use `?` for that
- `Apple <: Comparable<? super Apple>` right?
- `static <T extends Comparable<? super T>> T max(List<T> list)`

Q3di

- Target typing: `T <: Fruit`
- Argument typing: `List<Apple> <: List<T>` $\Longrightarrow$ `T = Apple`
- Bounds on T: `T <: Comparable<? super T>`
- T <: Fruit holds
- Also Apple <: Comparable<? super Apple> because Apple is a Comparable<Fruit>

Q3dii

- Target typing: `T <: Apple`
- Argument typing: `List<Apple> <: List<T>` $\implies$ `T = Apple`
- Bounds on T: `T <: Comparable<? super T>`
- T <: Fruit holds
- Also Apple <: Comparable<? super Apple> because Apple is a Comparable<Fruit>

Q3

- Note that `Apple <: Fruit <: Comparable<Fruit> <: Comparable<? super Fruit> <: Comparable<? super Apple>`