CS2030S Recitation

Week 5: Problem Set 4

brian

2025-09-21

National University of Singapore

Recap

Recap: Variance

- Covariant
  - `T <: S` $\Rightarrow$ `C(T) <: C(S)`
- Contravariant
  - `T <: S` $\Rightarrow$ `C(S) <: C(T)`
- Invariant
  - Neither covariant nor contravariant

Recap: Generics

- A type abstraction where you abstract on the type within a class
- `List<T>` can work without knowing what `T`
- Use bounds to expose methods (upper-bound)
- Java generics are invariant

Recap: Wild cards

- A substitute for any type
- Can be upper bounded (`? extends T`)
  - `?` can be `T` and its subtypes
  - Gives rise to covariance
- Can be lower bounded (`? super T`)
  - `?` can be `T` and its supertypes
  - Gives rise to contravariance

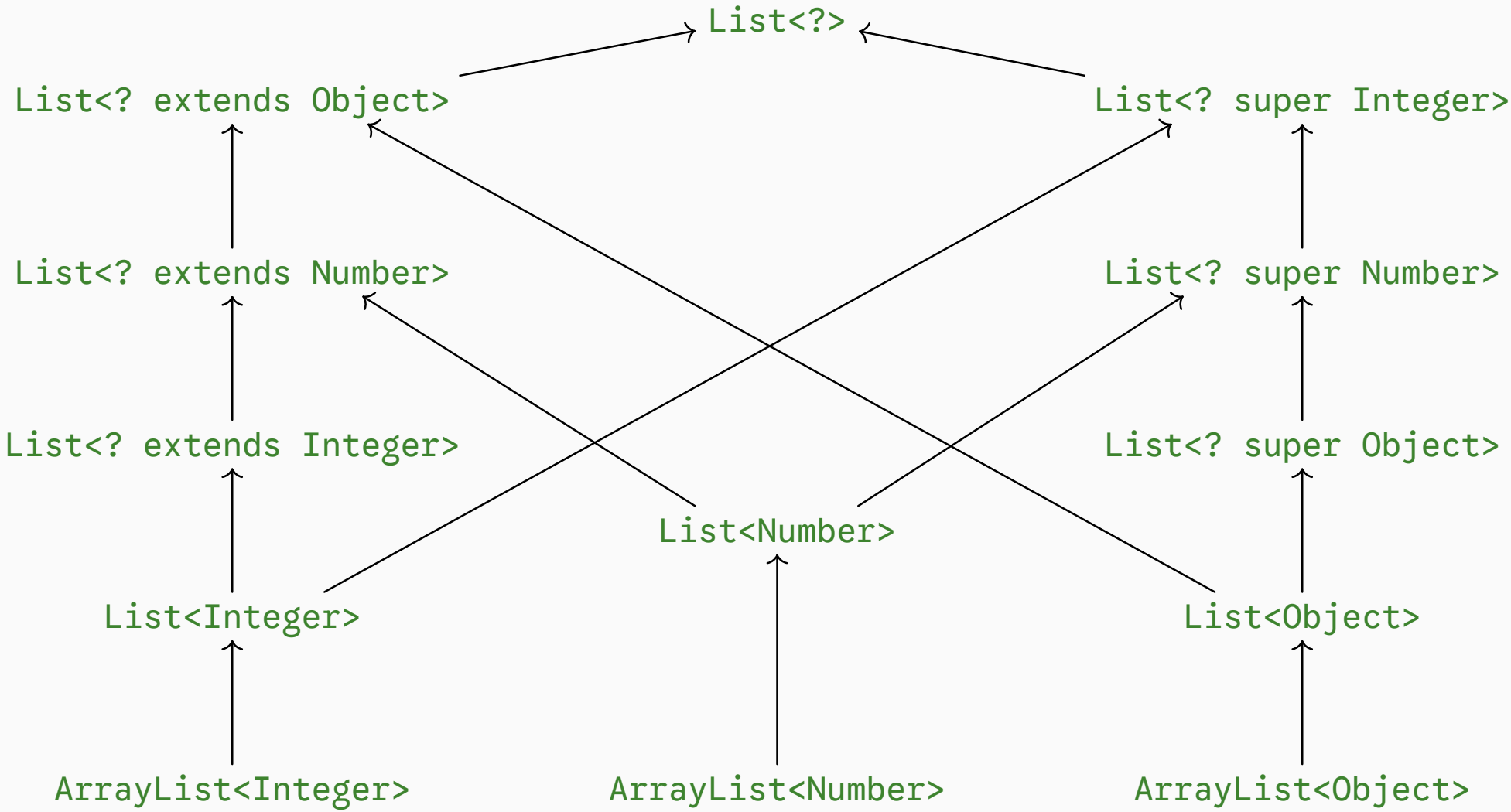
Recap: Wildcard vs type variable

- What is the difference between using `T` vs `?`
  - `?` if you don't know the type and you don't care
    - you can't even add `Object` to `List<?>` only `null`
  - `T` when you define a class that should work with any type

Q1

- Determine subtyping for all the mixtures of bounded wildcards
- `Integer <: Number <: Object . ArrayList<T>`
- Draw `S` $\rightarrow$ T, if `S <: T` (Hasse diagram)
- can omit transitive subtyping

Q1


List<?>
List<? extends Object>
List<? super Integer>
List<? extends Number>
List<? super Number>
List<? extends Integer>
List<? super Object>
List<Number>
List<Integer>
List<Object>
ArrayList<Integer>
ArrayList<Number>
ArrayList<Object>

Recap: PECS

- Think from the perspective of the container (`List<T>`, `Box<T>`)
  - What should be accepted **with respect to** whatever `T` is set to
- `P`roducer `E`xtends
  - If the container is producing at least `T` you can also get something that produces `SubT`
- `C`onsumer `S`uper
  - If the container is consuming at most `T` you can always store `T` in a container of `SuperT`

Recap: PECS Example

```java
public class Example<T> {
  void foo(Producer<? extends T> p, Consumer<? super T> c) {
    T t = p.produce(); // Producer<SubT>
    c.consume(t); // Consumer<SuperT> assigning subtype to a supertype
  }
}
```

Recap: Type Inference

- Find constraints (Assuming type variable `T`)
  - Argument type: Is there wildcards used for the `T` in the argument
  - Target type: Is `T` going to be bound to some type
  - Bounds on `T`: Does `T` extend/super something
- Solve the constraints
  - Ignore the subclasses not specified in the constraints
  - Solution may be a superclass of the types in the constraints

Q2a

`Apple <: Fruit <: Comparable<Fruit>`

```
1 static <T extends Comparable<T>> T max(List<T> list) {
2   T max = list.get(0);
3   if (list.get(1).compareTo(max) > 0) {
4     return list.get(1);
5   }
6   return max;
7 }
```

```
1 List<Fruit> fruits = List.of(new Fruit(), new Apple());
2 List<Apple> apples = List.of(new Apple(), new Apple());
```

What is the inferred type of `T` for

```
1 Fruit f = max(fruits);
```

Q2a

- Target: T <: Fruit
- Arugment: List<Fruit> <: List<T> $\Rightarrow$ T = Fruit
- Bounds: T <: Comparable<T>
- T would be **Fruit**

Q2bi

`Apple <: Fruit <: Comparable<Fruit>`

```
1 static <T extends Comparable<T>> T max(List<T> list) {
2   T max = list.get(0);
3   if (list.get(1).compareTo(max) > 0) {
4     return list.get(1);
5   }
6   return max;
7 }
```

```
1 List<Fruit> fruits = List.of(new Fruit(), new Apple());
2 List<Apple> apples = List.of(new Apple(), new Apple());
```

Why is there a compilation error for

```
1 Fruit f = max(apples);
```

Q2bi

- Target: T <: Fruit
- Argument: List<Apple> <: List<T> ⇒ T = Apple
- Bounds: T <: Comparable<T>
- If T is `Apple`, is it a subtype of `Comparable<T>`?
  - No
  - Cannot solve all constraints so compile error

```
Apple <: Fruit <: Comparable<Fruit>
```

```
1 static <T extends Comparable<T>> T max(List<T> list) {
2   T max = list.get(0);
3   if (list.get(1).compareTo(max) > 0) {
4     return list.get(1);
5   }
6   return max;
7 }
```

```
1 List<Fruit> fruits = List.of(new Fruit(), new Apple());
2 List<Apple> apples = List.of(new Apple(), new Apple());
```

Why is there a compilation error for

```
1 Apple a = max(apples);
```

Q2bii

- Target: T <: Apple
- Argument: List<Apple> <: List<T> ⇒ T = Apple
- Bounds: T <: Comparable<T>
- If T is `Apple`, is it a subtype of `Comparable<T>`?
  - No
  - Cannot solve all constraints so compile error

Q2biii

`Apple <: Fruit <: Comparable<Fruit>`

```
1 static <T extends Comparable<T>> T max(List<T> list) {
2   T max = list.get(0);
3   if (list.get(1).compareTo(max) > 0) {
4     return list.get(1);
5   }
6   return max;
7 }
```

```
1 List<Fruit> fruits = List.of(new Fruit(), new Apple());
2 List<Apple> apples = List.of(new Apple(), new Apple());
```

Why is there a compilation error for

```
1 Apple a = max(fruits);
```

Q2biii

- Target: T <: Apple
- Argument: List<Fruit> <: List<T> ⇒ T = Fruit
- Bounds: T <: Comparable<T>
- If T is `Fruit`, is it a subtype of `Apple`?
  - No
  - Cannot solve all constraints so compile error

Q2c

Fix the header of `max` so that `i` and `ii` work

- What was the main issue with `i` and `ii`
  - `Apple </: Comparable<T>`
- How can we solve this?
  - Wildcards?
  - Extend or super?

Q2c

- We should put `<T extends Comparable<? super T>>`
  - `compareTo` takes in a `T` so it is a Consumer

```
1 static <T extends Comparable<? super T>> T max(List<T> list) {
2   T max = list.get(0);
3   if (list.get(1).compareTo(max) > 0) {
4     return list.get(1);
5   }
6   return max;
7 }
```

Q2d

What is the new inferred type for `T` in `i` and `ii`

- `(i)`
  - Target: T <: Fruit
  - Argument: List<Apple> <: List<T> ⇒ T = Apple
  - Bounds: T <: Comparable<? super T>
  - `T` is `Apple`
- `(ii)`
  - Target: T <: Apple
  - Argument: List<Apple> <: List<T> ⇒ T = Apple
  - Bounds: T <: Comparable<? super T>
  - `T` is `Apple`

Q2d

- `(iii)`
  - Target: T <: Apple
  - Argument: List<Fruit> <: List<T> ⇒ T = Fruit
  - Bounds: T <: Comparable<? super T>
  - `Fruit </: Apple` so still compilation error
  - anyway these stuff intuitively does not make sense
  - If you have a list that can contains all kinds of fruits, how can you know you will get an apple

The End

bye!