ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Exercise 10

Parametric Polymorphism

November 27, 2020

Task 1

(from a previous exam)

Consider the following Java code:

```
class Car<T> {
    private List<? extends T> passengers;

    public Car(List<? extends T> passengers) {
        this.passengers = passengers;
    }
}
```

Remember that `List<E>` in Java contains a method `addAll` with the following signature:

```
boolean addAll(Collection<? extends E> c)
```

The method `addAll` adds all elements of the given collection `c` to the receiver list and returns `true` if the receiver list was modified.

A) We want to add a method to `Car<T>` that takes a list of passengers `p` to board the car. After the method is executed, the field `passengers` should refer to a list containing both the previous elements and the elements of `p`.

```
public void board(List<? extends T> p)
```

The following implementation is rejected by the compiler:

```
public void board(List<? extends T> p) {
    this.passengers.addAll(p);
}
```

Assume the body of `board` is exempted from the type checker. Provide code that calls `board` and inserts a string into a list of integers. Your code has to type-check.

solution

```
List<Integer> list1 = new LinkedList<Integer>();
Car<Object>   car   = new Car<Object>(list1);
List<String>  list2 = new LinkedList<String>();
list2.add("");
car.board(list2);
```

B) Give a new implementation of `board` (without modifying its signature) that implements the expected functionality and type-checks.

solution

```
public void board(List<? extends T> p) {
    List<T> b = new LinkedList<T>();
    b.addAll(this.passengers);
    b.addAll(p);
    this.passengers = b;
}
```

C) We now want to add a method to class `Car<T>` that transfers all passengers from this car to a given car. Fill in the blank to achieve the least restrictive but correct implementation.

```
public void transferPassengers(Car<__________> other) {
    other.board(this.passengers);
}
```

solution

```
? super T
```

Task 2

Consider the following Java method:

```
public void add(Object value, List<?> list) {
   list.add(value);
}
```

The Java compiler rejects this program, with the following message:

```
The method add(capture#1-of ?) in the type List<capture#1-of ?> is
not applicable for the arguments (Object)
```

A) Explain why we obtain such an error.

solution

We do not have any relation between the wildcard of `List`, and the types of the values that we are going to store.

B) Fix the program by using a generic type for the parameter of method `add` and constraining the wildcard appropriately.

solution

```
public <V> void add(V value, List<? super V> list) {
   list.add(value);
}
```

We have to use a lower bound constraint because we want the argument of `list.add` to be a supertype of `V`, otherwise we cannot pass it as a parameter.

C) We can use the following alternative signature for `add`:

```
public <V> void add(V value, List<V> list)
```

Is this solution more restricted than the one obtained using the wildcard?

solution

This method has exactly the same constraints as the ones obtained using a wildcard. In fact, the type of `value` can be a subtype of the type parameter of `list`, since it is a method argument. In practice, this means that the generic type of `list` is supertype of the type of `value`. For instance, consider the following program:

```
List<Object> list = new ArrayList<Object>();
add("x", list);
```

This program is accepted because `String <: Object`, thus `V=Object` is inferred by the type checker.

D) Consider the following methods:

```
public <V> void addAllX(List<V> v, List<? super V> l) {
   for(V el : v) l.add(el);
}
public <V> void addAllY(List<V> v, List<V> l) {
   for(V el : v) l.add(el);
}
```

Method `addAllX` is less restrictive than `addAllY`. Provide an example to prove this claim.

solution

```
List<String> listStr = new ArrayList<String>();
List<Object> listObj = new ArrayList<Object>();
addAllX(listStr, listObj);
addAllY(listStr, listObj);
```

The call to `addAllX` is accepted by the compiler, while the one to `addAllY` is rejected, since it requires that the parametric type of `List` is exactly `String`. This happens because the type parameters are invariant in Java, so `V` has to be `String`, but the generic type of `listObj` is `Object`.

Task 3

Consider the following class relations and the definition of method `foo`

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

B foo(List<? super B> list1, List<? extends B> list2) {
   list1.add(0, list2.get(0));
   return list2.get(0);
}
```

in which the signatures of the methods of `List<T>` are

```
public void add(int index, T value) {...}
public T get(int index) {...}
```

Can the method body be typechecked with respect to the method signature?

solution

The typechecker knows that

`∃ T1 >: B ∧ ∃ T2 <: B`
and has to prove that
`T2 <: T1 // list1.add(0, list2.get(0))`
AND
`T2 <: B // return list2.get(0);`

The assumptions are generated from the method signature, while the proof obligations are generated from the method body. This implication holds since `T2 <: T1` because of the transitive property of (`<:`), and `T2 <: B` directly from the hypothesis. Thus, the program typechecks.

Task 4 (from a previous exam)

A) Recall the Java interface `Comparable<T>` that was shown in the lecture:

```
public interface Comparable<T> {
    public int compareTo(T other);
}
```

The method `compareTo` returns $\begin{cases} 1 & \text{if } \texttt{this} \text{ is greater than } \texttt{other} \\ 0 & \text{if } \texttt{this} \text{ is equal to } \texttt{other} \\ -1 & \text{if } \texttt{this} \text{ is less than } \texttt{other} \end{cases}$

Suppose we want to turn `Comparable` into an abstract class with an additional helper method `greaterThan`, that returns true if and only if `this` is greater than `other`.

Assume the following implementation:

```
public abstract class Comparable<T> {
    public abstract int compareTo(T other);

    public boolean greaterThan(T other) {
        return other.compareTo(this) < 0;
    }
}
```

A.1) Why does this implementation not type check?

solution

`T` requires an upper bound that provides `compareTo`.

A.2) Fix the type error by changing only the body of `greaterThan`, while preserving the intended semantics of the method.

solution

```
return this.compareTo(other) > 0;
```

A.3) Fix the type error by changing only the class signature and the signature of the method `compareTo`.

solution

Let the class signature be `Comparable<T extends Comparable<T>>` and the signature of `compareTo` be `compareTo(Comparable<T> other)`.

B) Suppose we have the following class:

```
class A<X                    ,Y                    > {
    X a;
    Y b;
}
```

Consider a variable `v` whose type is `A<S,T>` where `S` and `T` satisfy the type bounds that you have to insert above. Your type bounds have to guarantee that for all sequences of `a` and `b` accesses on `v` (e.g., `v.b.a.b.a.a.b.b`) the following two properties hold:

- The static type of a sequence ending in `a` is `S`.
- The static type of a sequence ending in `b` is `T`.

solution

```
class A<X extends A<X,Y>,Y extends A<X,Y>> {
    X a;
    Y b;
}
```

Task 5

Consider the following Scala classes:

```
class A
class B extends A
class P1[+T]
class P2[T <: A]
```

What are the possible instantiations of `P1` and `P2`? What is the difference between `P1[A]` and `P2[A]` from the perspective of a client? Provide an example to show which class is more restrictive.

solution

Class `P1` can be instantiated with any type, while `P2` has to be instantiated with subtypes of `A`.

```
val x : P1[AnyRef] //correct
val y : P2[AnyRef] //wrong: AnyRef is not a subtype of A
```

Furthermore, class `P1` is covariant in its argument:

```
val x : P1[A] = new P1[B] //correct
val y : P2[A] = new P2[B] //wrong: found P2[B], required P2[A]
```

Task 6

Consider the following Scala definitions:

```
class PartialFunction[-F, +T]

class A
class B extends A
class C extends B

class X { def foo(): PartialFunction[B, B] }
```

Which of the following methods would be a valid override of the above method `foo`?

(a) `override def foo(): PartialFunction[A, A]`

(b) **CORRECT:** `override def foo(): PartialFunction[A, C]`

(c) `override def foo(): PartialFunction[C, A]`

(d) `override def foo(): PartialFunction[C, C]`

(e) None of the above

Task 7

(from a previous exam)

A) Suppose we have a simple list interface in Java:

```
public interface List<T> {
    public int length();
    public T get(int i);
    public void add(T element);
}
```

We want to implement a class that concatenates two lists while inserting a separator of some type `A` between the two lists:

```
public class Concatenator<A> {
    public void concatenate(A separator, List<A> from, List<A> to) {
        to.add(separator);
        for (int i = 0; i < from.length(); i++) {
            to.add(from.get(i));
        }
    }
}
```

We are unsatisfied with our signature of the `concatenate` method because it is too restrictive. In the following subtasks, we change the signature of the `concatenate` method, without changing its body, while making sure that the body still type-checks and that only instances of subtypes of `A` can be passed as separators.

We will try to make the signature less restrictive in the following sense. A signature s_1 of `concatenate` is *less restrictive* than another signature s_2 if the following holds: for all types T_1, T_2, T_3, if arguments of static type T_1`,List<`T_2`>,List<`T_3`>` are accepted by s_2, they are also accepted by s_1, but the same property does not hold in the opposite direction.

Do not use raw types (e.g. do not use `List` without a type variable). Do not use more than one upper bound per generic variable (e.g. do not use `X extends A & B`).

A.1) Provide the *least restrictive* signature using wildcards but no additional type parameters.

solution

```
public void concatenate(A separator,
                        List<? extends A> from,
                        List<? super A> to)
```

A.2) Provide a signature that is *less restrictive* than the original signature, without using wildcards, but with one extra type parameter to `concatenate`.

solution

Solution 1:

```
public <B extends A> void concatenate(A separator,
                                      List<B> from,
                                      List<A> to)
```

or Solution 2:

```
public <B extends A> void concatenate(B separator,
                                      List<B> from,
                                      List<B> to)
```

or Solution 3:

```
public <B extends A> void concatenate(B separator,
                                      List<B> from,
                                      List<A> to)
```

A.3) Provide the *least restrictive* signature without using wildcards, but using any number of type parameters to `concatenate`.

solution

```
public <C extends A, B extends C> void concatenate(C separator,
                                                   List<B> from,
                                                   List<C> to)
```

B) Provide the *least restrictive* signature without using wildcards or additional type parameters. For this subtask, assume that Java provides the variance modifiers known from Scala. Besides modifying the signature of `concatenate`, you may add interfaces and let existing interfaces implement them.

solution

```
public interface GetList<+A> {
    public int length();
    public A get(int i);
}
public interface AddList<-A> {
    public void add(A element);
}
public interface List<A> extends AddList<A>, GetList<A> {
    //...
}
public void concatenate(A separator, GetList<A> from, AddList<A> to) {
    //...
}
```

C) In each the following subtasks (C.1-C.3), compare the restrictiveness of the given pair of signatures from the previous subtasks (A.1-B). If one signature is less restrictive than the other, provide an example of static types which are accepted by one but not by the other signature.

For illustration, you can assume that we have three classes `X,Y,Z` with `X <: Y <: Z`, and we are calling `concatenate` on a class of type `Concatenator<Y>`. An example which shows differing restrictiveness then consists of a triple $T_1, T_2, T_3 \in \{\texttt{X}, \texttt{Y}, \texttt{Z}\}$, such that arguments of types T_1,`List<`T_2`>`,`List<`T_3`>` are accepted by one, but not by the other signature.

C.1) Compare solutions A.1 and A.3.

solution

A.3 is incomparable to A.1:

- We can call `concatenate(Y, List<X>, List<Z>)` in solution A.1, but not in solution A.3.
- We can call `concatenate(X, List<X>, List<X>)` in solution A.3, but not in solution A.1.

C.2) Compare solutions A.2 and A.3.

solution

- For Solution 1 and 3 in A.2: A.2 is strictly more restrictive than A.3: We can call `concatenate(X, List<X>, List<X>)` in solution A.3, but not in solution A.2.
- For Solution 2 in A.2: A.2 is strictly more restrictive than A.3: We can call `concatenate(Y, List<X>, List<Y>)` in solution A.3, but not in solution A.2.

C.3) Compare solutions A.1 and B.

solution

A.1 and B have the same restrictiveness.