ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Exercise 9
Parametric Polymorphism
November 26, 2021

Task 1

(from a previous exam)

Consider the following Java code:

```java
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:

```java
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`.

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

The following implementation is rejected by the compiler:

```java
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.

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

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.

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

Task 2

Consider the following Java method:

```java
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.

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

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

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

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

D) Consider the following methods:

```java
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);
}
```

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

Task 3

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

```java
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:

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

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

Task 4

Consider the following Scala classes:

```scala
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.

Task 5

Consider the following Scala definitions:

```scala
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) `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 6

(from a previous exam)

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

```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:

```java
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.

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

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

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.

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 $T_1, T_2, T_3 \in \{$`X,Y,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.

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

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