

Exercise 8

Parametric Polymorphism, Type Erasure, and Templates

November 25, 2022

Tasks covered in class

Tasks 1-7 will be covered in the exercise session. The remaining tasks are material for self-study.

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.

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.

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

Task 2

Consider the following class relations and the definition of the 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?

Task 3

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) `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 4

(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 $T_1, \text{List}\langle T_2 \rangle, \text{List}\langle T_3 \rangle$ 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 $T_1, \text{List}\langle T_2 \rangle, \text{List}\langle T_3 \rangle$ 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.

Task 5

Consider the following Java method:

```

String concatenate(List<?> list) {
    String result = "";
    String separator = "";
    if (list instanceof List<String>) {
        result = "String:";
        separator = " ";
    }
    else if (list instanceof List<Integer>) {
        result = "Integers:";
        separator = "+";
    }
}

```

```

    }
    for(Object el : list)
        result = result + separator + el.toString();
    return result;
}

```

A) This program is rejected by the Java compiler. Why?

B) Using the advice given by the Eclipse Java compiler (replace `List<...>` with `List<?>`), rewrite and compile the program. What are the results of executing the method passing each of the following:

- A list of strings containing only one element "word"?
- A list of Integers containing only one element `Integer(1)`?
- A list of Objects containing only one element (initialized by `new Object()`)?

C) Is this behaviour consistent with what you would expect from the initial program? If not, how can you fix it?

D) What would happen if you tried to implement the different cases using method overloading instead of just one method? Why is this the case?

E) What happens if you compile and execute the initial program in C# ? Why? (Assume that we replace the wildcard by a method type parameter `T` to make it work in C#.)

Task 6

From a previous exam

Consider the following Java program, which compiles correctly and makes use of generics:

```

1  class Animal {}
2  class Mammal extends Animal {}
3  class Tiger extends Mammal {}
4
5  class Ship<T extends Animal> {
6      public T content;
7  }
8
9  class Cage<T extends Mammal> {
10     public T content;
11
12     void takeFromShip(Ship<T> other) {
13         this.content = other.content;
14         other.content = null;
15     }
16 }
17
18 class Zoo<T extends Mammal> {
19     void swapTigers(Ship<T> mammalShip, Cage<Tiger> tigerCage) {
20         Tiger tiger = tigerCage.content;
21         Cage<Tiger> tmpCage = new Cage<Tiger>();
22         tmpCage.takeFromShip((Ship<Tiger>) mammalShip);
23         mammalShip.content = (T) tiger;
24         tigerCage.content = tmpCage.content;
25     }
26 }

```

A) List all the typecasts that the virtual machine will perform at runtime, when executing the methods `takeFromShip` and `swapTigers`. For each cast, write at which line number in the original program it is performed, and what expression is cast to which type. Do not optimize away casts that are statically known to succeed.

B) For each of the following two methods (from **B.1** and **B.2**), write if they would compile without errors if added to the class `Cage`. If they do not compile, briefly explain why.

B.1

```
Cage<Tiger>[] getTigers(int number) {
    Cage<Tiger>[] cages = new Cage<Tiger>[number];
    for (int i = 0; i < number; i++) {
        cages[i] = new Cage<Tiger>();
        cages[i].content = new Tiger();
    }
    return cages;
}
```

B.2

```
int numCageFields() {
    Class cl = Cage<Animal>.class;
    return cl.getFields().length;
}
```

Task 7

A C++ template class can inherit from its template argument:

```
template <typename T>
class SomeClass : public T { ... }
```

A) Using this technique and given the following class definition

```
class Cell {
public:
    virtual void setVal(int x) { x_ = x; }
    virtual int value() { return x_; }
private:
    int x_;
}
```

write two template classes that can be used as “mixins” for the class `Cell`:

- `Doubling` - doubles the value stored in the cell.
- `Counting` - counts the number of times the value of the cell was read.

Do not use multiple inheritance. It should be possible to use the classes like this:

```
auto c = new Doubling<Counting<Cell>>(); // instantiation
c->setVal(5);
c->value(); // returns 10
c->numRead(); // returns 1
```

B) Describe how the instantiation above will look like.

C) How does this concept of mixins in C++ differ from Scala traits?

Tasks not covered in class

Task 8

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.

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`:

```
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:

```
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 9

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.

Task 10

The type correctness of a C++ template class is checked only when the template is instantiated. This makes it difficult to develop templates modularly. We can try to make templates more modular by extending C++ with a new way to declare type arguments:

```
template<T s_extends SomeClass>
class TemplateClass {...}
```

Here T is the template argument and `SomeClass` is the name of a class which is an upper type bound for T . A template defined in this way may only be instantiated with a class T that is a *structural* subtype of `SomeClass`. Assume that the type checker checks such a template *definition* without having any concrete instantiation, under the assumption that T is a structural subtype of `SomeClass`.

This new feature is the only place where we introduce structural subtyping in C++, all other subtype relations in the language remain nominal as usual. Assume in general for any subtyping mode that method argument types are contravariant and method return types are covariant. Also, assume that all the methods are public and virtual.

A) Provide a declaration of the `Operation` class such that the class `Compose` can be type-checked before it is instantiated.

```
template<T s_extends Operation , U s_extends Operation>
class Compose : public Operation {
    public:
        T* t;
        U* u;
        int compute(int x) {
            return t->compute(u->compute(x));
        }
}
```

B) We also allow template parameters to occur as type arguments in upper bounds of the same template:

```
template<T s_extends Bound<T>>
class TemplateClass{...}
```

The above limits the possibilities for T to only structural subtypes of `Bound<T>`.

Consider the classes below:

```
class A : { void foo(A* a); };
class B : public A { B* bar(); };
class C : public B {};
```

```
template <class T>
class FOO {
    void foo(T* t){...}
};
```

```
template <T s_extends FOO<T>>
class X { ... };
```

```
template <class T>
class BAR {
    T* bar(){...}
};
```

```
template <T s_extends BAR<T>>
class Y { ... };
```

Which of the following instantiations typecheck:

X
X<C>
Y
Y<C>

Explain why each combination does or does not typecheck.

C) As a bound we also allow the template that is being declared:

```
template <T s_extends X<T>>
class X {
    int foo(T* t) {...}
}
```

Let the class A be:

```
class A {};
```

- Write an implementation of the body of the `foo` method of `X` such that `X` typechecks with the bound above (`T s_extends X<T>`) and also typechecks if the bound is changed to `T s_extends A`.
- Write an implementation of the body of the `foo` method of `X` such that `X` typechecks with the bound above (`T s_extends X<T>`), but does not typecheck if the bound is changed to `T s_extends A`.
- Write a class `B` that can be used to instantiate `X`.

D) A C++ template class can inherit from its template argument:

```
template <class T>
class Mixin : public T { ... }
```

Such a template is called a mixin. We want to use the newly introduced template bound feature `<T s_extends ...>` in order to create a mixin that is guaranteed only to override existing methods but not introduce new ones. Show how this can be done.