ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

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.

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 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?

solution

The typechecker knows that

$\exists$ `T1 >: B` $\wedge$ $\exists$ `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 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) **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 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, `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 $T_1, T_2, T_3 \in \{$`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.

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.

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?

solution

The Oracle and the Open JDK compilers both produce these short errors:

```
illegal generic type for instanceof
illegal generic type for instanceof
```

The Eclipse compiler tries to be more helpful:

```
Cannot perform instanceof check against parameterized type
List<String>. Use the form List<?> instead since further
generic type information will be erased at runtime

Cannot perform instanceof check against parameterized type
List<Integer>. Use the form List<?> instead since further
generic type information will be erased at runtime
```

This happens because of type erasure in Java.

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())?

solution

First of all, we follow the output of the compiler, and so we rewrite the method to:

```
String concatenate(List<?> list) {
   String result = "";
   String separator = "";
   if(list instanceof List<?>) {
      result = "String:";
      separator = " ";
   }
   else if(list instanceof List<?>) {
      result = "Integers:";
      separator = "+";
   }
   for(Object el : list)
      result = result + separator + el.toString();
   return result;
}
```

The Java compiler will compile this program without any warning. The output of the method is:

```
String: word
String: 1
String: java.lang.Object@3e25a5
```

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

solution

No, in the original program we expected:

```
String: word
Integers:+1
java.lang.Object@3e25a5
```

We can try to fix it in the following way:

```
String concatenate(List<?> list) {
   String result = "";
   String separator = "";
   if(list.size() >= 1 )
      if(list.get(0) instanceof String) {
         result = "Strings:";
         separator = " ";
      }
      else if(list.get(0) instanceof Integer) {
         result = "Integers:";
         separator = "+";
      }
   for(Object el : list)
      result = result + separator + el.toString();
   return result;
}
```

But this requires to have at least one element in the list. Moreover, there is no guarantee that if the first element is, for example, a String, that this is not a list of Objects. Therefore, an improved solution would be to iterate over all the elements of the list and to compute their smallest common supertype.

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?

solution

If we introduce separate methods which differ only by the generic types of their arguments, we get compile-time errors such as:

```
Method concatenate(List<? extends Object>) has the same
erasure concatenate(List<E>) as another method in type C
```

This restriction is imposed to ensure that when choosing which of the overloaded method definitions to call, we always have a "best fit". Java class files do however include generic versions of the method signatures in the class (to enable separate compilation and type-checking of generic code). For this reason, it might seem surprising that we cannot disambiguate between these different overloaded methods, since at compile-time the type information is all available. However, Java also supports raw types - versions of generic classes in which no type parameter is provided (e.g., `List` for a `List<X>` class). These are supported for backwards compatibility with pre-generics Java code. For this reason, we need to consider the possibility that a client calling our method provides an argument of raw type `List`. In this case, we would not be able to choose between our different method overloads.

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#.)

solution

The program is compiled and we obtain the expected results ("String: word", "Integers:+1", "..."), since in C# there is no type erasure and the information about generics is preserved at runtime.

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.

solution

- At line 13: `(Mammal) other.content`
- At line 20: `(Tiger) tigerCage.content`
- At line 22 `(Ship) mammalShip`
- At line 23: `(Mammal) tiger`

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;
}
```

solution

1. Compiler error: array of generic types not allowed.
2. Compiler error: class object of generic types not available, or the type parameter does not respect the constraint of class `Cage`.

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
```

solution

```
template <typename T> class Doubling : public T {
public:
   virtual void setVal(int x) override {
      T::setVal(x * 2);
   }
}
```

```
template <typename T> class Counting : public T {
public:
   virtual int value() override {
      ++numRead_;
      return T::value();
   }
   int numRead() { return numRead_; }
private:
   int numRead_;
}
```

B) Describe how the instantiation above will look like.

solution

When the mix-ins are instantiated the following two classes will be generated:

```
class Counting_Cell : public Cell {
public:
   virtual int value() override {
      ++numRead_;
      return Cell::value();
   }
   int numRead() { return numRead_; }
private:
   int numRead_;
}

class Doubling_Counting_Cell : public Counting_Cell {
public:
   virtual void setVal(int x) override {
      Counting_Cell::setVal(x * 2);
   }
}
```

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

solution

While this concept is similar to Scala traits there are some notable differences. In Scala it is possible to mix any number of traits in a class and use this in any location of the code that requires the same class and a subset of the traits:

```
var x = new X with A with B with C with D
var x1: (X) = x // OK
var x2: (X with A) = x // OK
var x3: (X with B) = x // OK
var x4: (X with A with D with C) = x // OK
```

Using the proposed solution in C++ however is more restrictive, as there is no way to refer to the class `X` with arbitrary mix-ins:

```
auto x = new D<C<B<A<X>>>>();
X* x1 = x; // OK
A<X>* x2 = x; // OK
B<X>* x3 = x; // Does not compile
C<D<A<X>>>* x4 = x; // Does not compile
```

This is particularly important for traits that introduce new methods like `Counting.numRead()` since any client code that uses this new behavior would have to know exactly how the trait was mixed-in.

Another problem of the C++ solution is object construction. If the base class does not have a default constructor then the mix-ins should know to call the correct constructor and provide appropriate parameters. An alternative here is for the mixin to just inherit the base class constructors: `using T::T;` which will allow clients of the mixin to use all constructor available in the base class. This works fine if the state of the mixin can be initialized with default values.

A further difference to Scala is that in the C++ solution it is possible to include the same "trait" more than once:

```
auto x = new Doubling<Doubling<X>>();
x->setVal(5);
x->value(); // returns 20
```

An advantage of the C++ solution is that we do not need to declare the base class that the mix-ins extend. Thus it is possible to use them with different base classes as long as they have matching virtual methods.

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.

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 the 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 one 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 a 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);
}
```

The 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 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.

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 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));
      }
}
```

solution

```
class Operation {
   int compute(int 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<B>
X<C>
```

```
Y<B>
Y<C>
```

Explain why each combination does or does not typecheck.

solution

`X` can be instantiated with both classes:

- `B`: `B.foo(A*)` overrides `FOO<B>.foo(B*)`
- `C`: `C.foo(A*)` overrides `FOO<C>.foo(C*)`

`Y` can be instantiated only with `B`:

- `B`: `B* B.bar()` overrides `B* BAR<B>.bar()`
- `C`: `B* C.bar()` does not override `C* BAR<C>.bar()`, therefore `C` is not a structural subtype of `BAR<C>`

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`.

solution

- `int foo(T* t) { return t ? 1 : 0; }`
- `int foo(T* t) { return t->foo(t); }`
- `class B { int foo(B* b) { return 0; } }`

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.

solution

```
template <T s_extends Mixin<T>>
class Mixin : public T
```

Here we restrict the base class `T` to be a structural subtype of `Mixin<T>`. Thus all admissible base classes `T`, have at least the methods that `Mixin<T>` defines in its body.

If the mixin tried to add a new method, this would fail. For example:

```
template <T s_extends Mixin<T>> class Mixin : public T {
  public: int foo(int x) { return x + T::foo(x); } // overriden method
          int bar(int y) { return 2 + y; } // newly-added method
}

class A { public: int foo(int x) { return 2 * x; } }
```

The instantiation `Mixin<A>` fails, because `A` is not a structural subtype of `Mixin<A>`, since `A` does not have the `bar` method.