ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Exercise 8

Parametric polymorphism

November 14, 2014

Task 1

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 Java compiler. Why?

solution

The Oracle and the Open JDK compilers both produces 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 Java compiler, 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 obviously:

```
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 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.

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?

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 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 of the ones obtained using a wildcard. In fact, the type of `value` can be a subtype of the generic type 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 =...
add("x", list);
```

This program is accepted because strings are subtype of objects, 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> list = new ArrayList();
List<Object> list2 = new ArrayList();
addAllX(list, list2);
addAllY(list, list2);
```

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 of invariance on type parameters in Java, so `V` has to be `String`, but the generic type of `list2` is `Object`.

Task 3

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 4

Consider the following Scala code:

```
class A[-T]
class B[ ... T] {
   def m(in : A[T]) : Int = {...}
}
```

We want to annotate the generic type of `B`. If we use a covariant or a contravariant annotation for the generic type parameter of `B`, what would that annotation be? Why? Justify your answer with an example.

solution

The only annotation that may be used here is + (covariant). Subtypes of `B` may override the `m` method. Due to contravariance of method arguments, valid subtypes of `B` may use a supertype of `A[T]` for the argument `in`. Due to the declared contravariance of `A[T]`, supertypes of `A[T]` are classes `A[X]` where `X <: T`. Thus, `B[X] <: B[T]` if `X <: T` and we must use a covariant annotation: `B[+T]`.

If we use contravariance, then the following code will break:

```
class A[-T] {
  def len(t:T) : Int = 1
}

class AStr extends A[String] {
   override def len(s:String) : Int = { s.length() }
}

class B[-T]
{
   def m(x:A[T]) : Int = 2
   // The compiler gives this error message on the previous line:
   // contravariant type T occurs in covariant position in type
```

```
   // A[T] of value x
}

class BAnyRef extends B[AnyRef]
{
   override def m(x:A[AnyRef]) : Int = { x.len(new Integer(3))} }

// Client
class F {
   def f() {
      val a = new AStr
      val b = new BAnyRef
      b.m(a) // If the code would compile, here we would try to
             // call length() on an Integer object
   }
}
```

Task 5

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 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 they have matching virtual methods.

D) Can the code above be implemented using Java generics? If yes, show how. If no, explain why not.

solution

No, it is not possible to implement this with generics. The core reason for this is erasure. In Java each class must have a known supertype. However if we could translate the code above to Java and apply erasure, it will turn out that the supertype of `Doubling` and `Counting` is `Object` which is clearly not what we want.

E) What if we used C# instead of Java, does anything change?

solution

The code cannot be implemented using C# generics either - the standard explicitly forbids a generic class to inherit directly from a type argument. Although the type argument would be known at run-time and it is theoretically possible to allow inheriting from it, that would have complicated and slowed down the handling of generics by the C# virtual machine. The reason is that unlike C++, in C# only a single class would be generated for all possible type arguments and a lot of dynamic checks and method call adjustments would be required to make this work. Thus in this case the designers of C# chose safety and efficiency at the expense of expressiveness.

Task 6 Modular templates

In the C++ code throughout this task assume that all methods are `public` and `virtual`. In your answers it is not important to get the syntax of C++ exactly right. However, make sure that what you write is clear and unambiguous.

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.

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*); };
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: not a subtype of `BAR<C>` because `B* C.bar()` does not override `C* BAR<C>.bar()`

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 not if the bound is changed to `T s_extends A`.
- Write a class B that can be used to instantiate X assuming the original definition of X.

solution

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

D) As we have seen in the exercises, 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 we tried to use the mixin with a base type that is not a structural subtype, 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); }
}

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

class AwithMixin : public Mixin<A>{}; // OK
class BwithMixin : public Mixin<B>{}; // Fails.
```

The last line fails, because `B` is not a structural subtype of `Mixin<B>` since `B` has no `foo` method.