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?

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())?

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?

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.

B) Fix the program by using a generic type for the parameter of 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);
}
```

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

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.

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 to `B`, what would that annotation be? Why? Justify your answer with an example.

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
```

B) Describe how the instantiation above will look like.

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

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

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

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));
      }
}
```

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.

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`.

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.