Concepts of Object-Oriented Programming
AS 2021

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Exercise 7

Linearization and Bytecode Verification

November 12, 2021

Task 1

Consider the following declarations in Scala:

```
class C
trait T extends C
trait U extends C
class D extends C
```

Find all the types that can be created with or without traits, as well as the subtype relations between them.

Task 2

Consider the following Scala code:

```
class Cell {
   private var x:Int = 0
   def get() = { x }
   def set(i:Int) = { x=i }
}

trait Doubling extends Cell {
   override def set(i:Int) = { super.set(2*i) }
}

trait Incrementing extends Cell {
   override def set(i:Int) = { super.set(i+1) }
}
```

A) What is the difference between the following objects?

```
val a = new Cell
val b = new Cell with Incrementing
val c = new Cell with Incrementing with Doubling
val d = new Cell with Doubling with Incrementing
```

B) We try to use the following code to implement a cell that stores the argument of the set method multiplied by four:

```
val e = new Cell with Doubling with Doubling
```

Why does it not work? What does it do? How can we make it work?

C) Find a modularity problem in the above, or a similar, situation. Hint: a client that is given a class `C` does not necessarily know if a trait `T` has been mixed in that class.

D) We propose the following solution to support traits together with behavioral subtyping: assume `C` is a class with specification `S`. Each time we create a new trait `T` that extends `C`, we must ensure that `C with T` also satisfies `S`. Show a counterexample that demonstrates that this approach does not work.

Task 3

(from a previous exam)

Consider the following Scala code:

```
class A           {          def bar() =                "" }
trait B extends A { override def bar() = super.bar() + "B" }
trait C extends B { override def bar() = super.bar() + "C" }
trait D extends B { override def bar() = super.bar() + "D" }

object Main {
  def main() { foo(new A with D with C with B) }
  def foo(x: A with D) { println(x.bar()) }
}
```

What would be the output of the call `Main.main()`?

(a) BDB

(b) BBDBC

(c) BBCBD

(d) DB

(e) BDC

(f) BCD

(g) None of the above

Task 4 (from a previous exam)

Consider the following Scala code, which compiles correctly and models some jobs a `Person` may have. To work as a `Lawyer` or as a `TaxiDriver`, one needs to have a valid license. This requirement can be expressed through *self type annotations* added to the traits `Lawyer` and `TaxiDriver` (as in the given code). These annotations are checked by the compiler and allow the traits `Lawyer` and `TaxiDriver` to be mixed only into subtypes of `PersonWithLicense`. Self type annotations enable code reuse without subtyping, that is, `Lawyer` and `TaxiDriver` $\not<:$ `PersonWithLicense`, but the methods of the class `PersonWithLicense` are available and can be overridden inside these two traits.

```
class Person { def work(): String = { return "working"; }}

class Student { def work(): String = { return "studying"; }}

class PersonWithLicense extends Person {
 def hasValidLicense(): Boolean = { return false; }
}

trait Gardener extends Person {
 override def work(): String = { return super.work() + " in the garden";}
}
```

```
trait Lawyer extends Person {
 this: PersonWithLicense => // self type annotation

 override def work(): String = {
   if(this.hasValidLicense()) return super.work() + " in court";
   return "not " + super.work();
 }

 override def hasValidLicense(): Boolean = { return true; }
}

trait TaxiDriver extends Person {
 this: PersonWithLicense => // self type annotation

 override def work(): String = { return super.work() + " in Zurich"; }
}
```

A) For each of the following two code fragments (**A.1** and **A.2**), if they compile, write the output of their execution. Otherwise, briefly explain why they are rejected by the compiler.

A.1

```
val lawyer: Lawyer = new PersonWithLicense with Lawyer with TaxiDriver;
println(lawyer.work());
```

A.2

```
val student: Gardener = new Student with Gardener;
println(student.work());
```

B) Add **one** method to any of the given classes or traits **except** `PersonWithLicense` (explicitly write to which one) and fill in the instantiation from the client code below, such that it compiles and when executed prints `not working in Zurich in the garden`. You are **not allowed** to directly return this string, to use reflection, to define new classes or traits, nor to modify the given code. If this is not possible, briefly explain why.

```
// Client code:
val person = new ____________________________________________________
println(person.work());
```

The following method should be added to: ____

Task 5

Consider a Java class `E`, which has a method `f` with the following signature: `void f();`

The method `f` has one local variable `v` and the following body:

```
0: iconst 5
1: istore 1
2: aload 0
3: astore 1
4: iload 1
5: iconst 1
6: iadd
7: istore 1
8: return
```

The maximal stack size is equal to 1. Can the provided bytecode be verified? If so then verify it, otherwise explain which line of code causes the problem and why.

Task 6

The Java bytecode verifier is more permissive than the Java type system. Provide a program that demonstrates this.

Task 7

Assume we have two Java classes `A` and `B`. Consider the following Java class `C`:

```
class C {
    void foo(A x) {
        int y = 7;
        this.bar(y, x);
    }

    B bar(int u, A v) {
       ...
    }
}
```

Assume that the method `foo` gets compiled into bytecode as follows:

```
0: iconst 7
1: istore 2
2: aload 0
3: aload 2
4: aload 1
5: invokevirtual C.bar.B(int,A)
```

Can this bytecode be verified? If so, what is the final state (after line 5)?