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.

solution

We can create the following types:

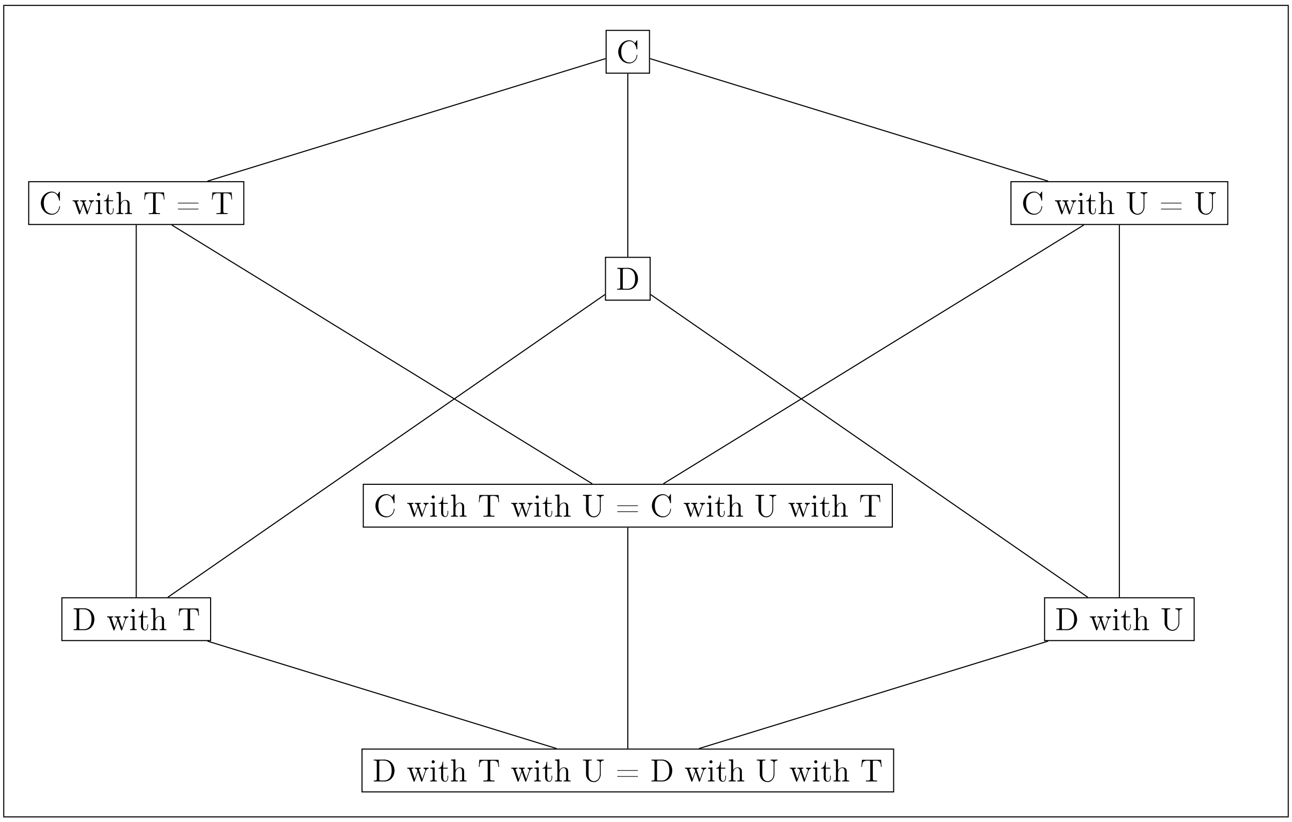
`C`, `D`, `T`, `U`,
`C with T` (same type as `T`, because `T extends C`),
`C with U` (same type as `U`, because `U extends C`),
`C with T with U` (same type as `C with U with T`),
`D with T`,
`D with U`,
`D with T with U` (same type as `D with U with T`).

The subtype relation is reflexive and transitive. Moreover, let X', Y' be the two base classes from which we derive X and Y by mixing in traits. Let A be the set of all traits mixed into the first class and B the set of all traits mixed into the second class. The rule is as follows:

$$X <: Y \text{ if and only if } X' <: Y' \text{ and } A \supseteq B.$$

Note: The above rule applies in our example, but it is not a general rule for subtyping in the presence of traits. Note that even if `D with T with U` and `D with U with T` are equivalent types (subtypes of each other), they can describe different behavior, causing subtle problems for behavioral subtyping.

We can also visualize the types and the subtype relations between them (the edges corresponding to reflexive and transitive subtype relations were omitted):



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
```

solution

Object `a` behaves like a normal cell. Object `b` is also a cell, but it increases the stored value by 1. The interesting difference is between `c` and `d`. They are both cells. They have mixed in exactly the same traits. However, calling `set(i)` has a different effect on them: it stores `2i+1` in the first one and `2(i+1)` in the second one.

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

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

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

solution

Trait `Doubling` will not get mixed in twice. Scala rejects this statically.

A possible attempt to bypass the problem is to create a new trait `Doubling2` that behaves exactly like `Doubling`:

```scala
trait Doubling2 extends Doubling
val e = new Cell with Doubling with Doubling2
```

The code passes through, but dynamically `e` behaves as if it were a `Cell with Doubling`. Scala lets the code go through, because `Doubling2` may introduce new functionalities, but does not include `Doubling` twice in the linearization.

Our last try (the one that works), is to create a whole new trait from scratch, without reusing existing code:

```scala
trait Doubling3 extends Cell {
   override def set(i:Int) = { super.set(2*i) }
}
val e = new Cell with Doubling with Doubling3
```

And now `e.set` quadruples its argument, as expected.

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.

solution

A problem is that a method that accepts `Cell with Doubling with Incrementing` as an argument could also be passed a class of the type `Cell with Incrementing with Doubling` - so what it can actually assume about its inputs is less than would be expected.

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.

solution

Consider the following example:

```scala
class Cell {
   var x:Int = 0
   // ensures x <= i + 1
   def set(i:Int) = { x=i }
}

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

trait Incrementing2 extends Cell {
   override def set(i:Int) = { super.set(i+1) }
}
```

Both `Cell with Incrementing` and `Cell with Incrementing2` are behavioral subtypes of `Cell`. But `Cell with Incrementing with Incrementing2` is not a behavioral subtype of `Cell`, as the following example shows:

```
val c: Cell = new Cell with Incrementing with Incrementing2
c.set(4)
assert(c.x <= 5) // fails, postcondition of Cell.set not fulfilled
```

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) **CORRECT:** BDC

(f) BCD

(g) None of the above

solution

The super calls are resolved based on the linear order.

`L(new A with D with C with B) = L(B), L(C), L(D), L(A)` (*)
`L(A) = A`
`L(D) = D, B, A`
`L(C) = C, B, A`
`L(B) = B, A`

We now substitute the linearizations of `A, D, C, B` in (*) (in this order) and make sure the same class/trait is not included twice:

`L(new A with D with C with B) = L(B), L(C), L(D), A`
`L(new A with D with C with B) = L(B), L(C), D, B, A`
`L(new A with D with C with B) = L(B), C, D, B, A`
`L(new A with D with C with B) = C, D, B, A`

The call `x.bar()` corresponds to `C.bar()`, as `C` is the first in the linear order.
`C.super().bar()` is `D.bar()`, as `D` follows after `C` in the linear order.

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.

```scala
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

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

> solution
>
> The code compiles: the traits `Lawyer` and `TaxiDriver` are mixed into a subtype of `PersonWithLicense`, as required by the self type annotation and `new PersonWithLicense with Lawyer with TaxiDriver` $<:$ `Lawyer`.
>
> The linearization of `new PersonWithLicense with Lawyer with TaxiDriver` is `TaxiDriver`, `Lawyer`, `PersonWithLicense`, `Person`. When executed, the code prints:
> working in court in Zurich

A.2

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

solution

The code does not compile, because `Student` is not a subclass of `Person` (the superclass of the trait `Gardener`)

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: ____________________

solution

```
// Client code:
val person = new PersonWithLicense with Lawyer with TaxiDriver with
   Gardener;
println(person.work());

trait TaxiDriver extends Person {
  ...
  // additional method:
  override def hasValidLicense(): Boolean = { return false; }
}
```

Note that if we try to add the method `hasValidLicense` to the trait `Gardener`, the client code does not compile, as `new PersonWithLicense with Lawyer with TaxiDriver with Gardener` inherits two methods with the same signature.

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.

solution

In the following, we try to verify the bytecode. `T` is an uninitialized register. A state is represented by a pair `(S,R)` where `S` describes the content of the stack and `R` describes the content of the registers.

```
// ([],[E,T]) -- initial state
iconst 5
// ([int],[E,T])
istore 1
// ([], [E,int])
aload 0
// ([E], [E,int])
astore 1
// ([], [E,E])
iload 1
// ERROR!
...
```

The error happens because `iload 1` expects that the local variable has the type integer, but its type is `E`.

Task 6

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

solution

Here is an example of such a program:

```
x = true;
x = 5;
```

The type of the variable can change in the bytecode but not in the source code.

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)?

solution

We assume that the maximal stack size is 3 and that $MR = 3$ (since we have three parameters/local variables): **this**, one argument (`x`), and one local variable (`y`). The initial state is `([], [C, A, T])`, where `C` is the type of **this**, `A` is the type of the argument `x` and the local variable `y` is uninitialized.

```
// ([],[C,A,T])
0: iconst 7
// ([int],[C,A,T])
1: istore 2
// ([],[C,A,int])
2: aload 0
// ([C],[C,A,int])
3: aload 2
// ERROR!
...
```

The error happens because `aload 2` expects that the local variable (from register 2) has a reference type, but its type is `int`.

Let's now assume that we correct the given bytecode, such that in line 3 we have `iload 2`. All the other instructions remain unchanged. We then obtain:

```
... // as before
3: iload 2
// ([int,C],[C,A,int])
4: aload 1
// ([A,int,C],[C,A,int])
5: invokevirtual C.bar.B(int,A)
// ([B],[C,A,int])
```

So the bytecode successfully verifies.