ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

# Exercise 7

## Bytecode Verification and Generics
## self-study exercise sheet

**NOTE: There will not be a regular exercise session on 6 November, and you will take the midterm exam instead. Therefore this exercise sheet will not be discussed in an exercise session. We publish it now together with the solution to allow you to better prepare for the midterm. If you have any questions regarding this sheet, please consult your assistant.**

## Task 1

The method `f` of class `E` has the following signature:

```
void f();
```

and one local variable `v`. The maximal stack size is equal to 1.

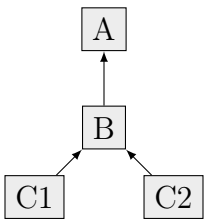
It has the following body:

```
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
```

Can the provided byte code be verified? If so then verify it, otherwise explain which line of the code causes the problem and why.

## Task 2

Consider the following type hierarchy:




Suppose that the method `f` of class `E` has the following signature:

```
A f(boolean b1, boolean b2);
```

and there are three local variables `x`, `y`, `z`. It is known that the initial state is:

```
([], [E,boolean,boolean,C1,C2,A])
```

The maximal stack size is equal to 1.

The method f has the following body:

```
 0: iload 1
 1: ifeq  22
 4: iload 2
 5: ifeq  12
 8: aload 3
 9: goto 14
12: aload 4
14: astore 3
15: aload 5
17: astore 4
19: goto 0
22: aload 3
23: areturn
```

Note: In this example, `ifeq x` pops an integer from the stack and jumps to line `x` if the integer is equal to zero.

**A)** Verify that the program is type safe.

**B)** Provide the minimal type information that enables verification of the bytecode without a fixpoint computation.

## Task 3

Consider the following Java code:

```
interface IFace {
   void m();
}
class Cl1 implements IFace {
   public void m() { System.out.println("Cl1.m"); }
}
class Cl2 implements IFace {
   public void m() { System.out.println("Cl2.m"); }
}
public class Test1 {
   public static void main( String[] args ) {
      xxx(true);
      xxx(false);
  }
   public static void xxx( boolean param ) {
      IFace iface = null;
      if( param ) { iface = new Cl1();}
      else { iface = new Cl2(); }
      iface.m();
   }
}
```

**A)** What type will be calculated for the variable `iface` of the method `xxx` during the bytecode verification?

**B)** When can we decide that `iface.m()` is safe to call? During bytecode verification, or execution?

**C)** What if `IFace` was a class instead of an interface? What if it was an abstract class?

## Task 4

The bytecode type inference algorithm rejects a verified program if there are different stack sizes for input values of a join point.

**A)** Provide a bytecode program that is rejected because of this limitation but that does not cause runtime errors.

**B)** Is it possible to construct a bytecode verification algorithm that avoids this limitation? If yes, then provide an updated algorithm. If no, then show that it can't be done.

**C)** How serious is this restriction from a pragmatic perspective?

## Task 5

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.

## Task 6

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.