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 11th of 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.

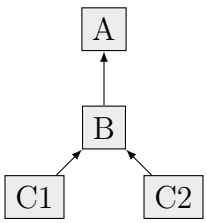
It has the following body:

```
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

Implement a list in Java or C# with two methods:

```
public void add(int i, Object el)
public Object get(int i)
```

Implement the list and discuss the advantages and the limitations of the three different approaches below.

A) Implement the list using only one class without generics.

B) Implement the list using one abstract class/interface and then (some) subclasses that implement it for different types.

C) Implement the list using generic types.

Task 6 Generics

Consider the following Java program, which is rejected by the Java compiler:

```
class Logger<T> {
    public void log(T t) {
        System.out.println(t.loggerString());
    }
}
```

A) If the code above were allowed to compile without errors, what could go wrong? To answer, write a brief code sample that uses `Logger` in a way which causes a failure at runtime.

B) How can we fix the class `Logger` so that it compiles, while preserving its functionality? You should not modify the method `log`, but otherwise can change or add any code. Your solution should include all details required to check that `Logger` is a valid Java class.

C) Assume that class `Logger` has been fixed to resolve the problem from point **A**. Let `A` and `B` be two classes such that `A` is a supertype of `B` and `Logger<A>` and `Logger<B>` are valid instantiations. Consider the following method:

```
void foo(Logger<A> logA) {
  Logger<B> logB = logA;
  logB.log(new B());
}
```

The Java compiler rejects this code. Is the code safe? That is, if it were allowed to compile, would it run without failure?

D) Suppose we relax the Java type system rules to allow contravariant generics.

- Will the method `foo` compile now?
- What are two situations that will require dynamic checks in order to enable contravariant generics in a language, without limiting what a developer can write in a generic class?