Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Exercise 9

Bytecode Verification and Parametric Polymorphism

November 20, 2020

Task 1

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 2

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)
```

Verify this bytecode using the type inference algorithm. What is the final state (after line 5)?

Task 3

(from a previous exam)

Assume two Java classes `A` and `B`, where `B` is a subclass of `A`. Consider the following bytecode:

```
0: aload 1
1: astore 2
2: goto 0
```

and assume that the input to the initial node of this code is `([],[A,A,B])`, where the first list indicates the content of the stack and the second list indicates the content of the registers.

After running the bytecode type inference algorithm, what is the inferred input to the initial node?

(a) `([],[A,A,A])`

(b) `([],[A,A,B])`

(c) `([],[A,B,B])`

(d) Nothing is inferred – the type inference does not terminate

(e) Nothing is inferred – the type inference rejects the program

Task 4

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 Test {
   public static void main(String[] args) {
      foo(true);
      foo(false);
   }
   public static void foo(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 `foo` during bytecode verification?

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

C) Would your answer from **B** be the same if `IFace` were a class instead of an interface? What if `IFace` were an abstract class?

Task 5

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

Task 6

In this task, you have to implement (using three different approaches) a list in Java that supports the following two methods (where `i` represents an index):

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

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 7

(from a previous exam)

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?