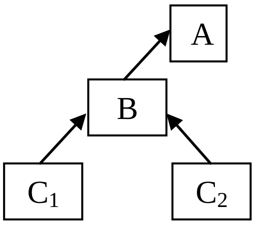
Exercise 6

Bytecode verification

November 5th

1) Consider the following type hierarchy:



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

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

and 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
```

- Verify that the program is type safe.
- Provide the minimal type information that enables verification of the bytecode without a fixpoint computation.

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

2) Consider the following Java classes

```
class C

class A extends C {
      void foo() {…}
}

class B extends C {
      void foo() {…}
}
```

Consider the following bytecode program (suppose that it is produced when compiling a method in class C):

```
0:     iconst_5
1:     ifeq 4
2:     new A
3:     goto 5
4:     new B
5:     invokevirtual A.foo()
```

`ifeq` jumps to the given index if the integer value at the top of the stack is equal to zero. `new D` creates a new instance of a class `D` and pushes a reference to this instance onto the operand stack. `invokevirtual D.m()` invokes method `m()` of class `D`. For this statement, suppose that the type checker checks that the type of the reference at the top of the stack is `D` or one of `D`’s subtypes.

- Show what will be computed by the type inference algorithm.
- The type checker will not validate this program. Why?
- Propose a modification of the verifier in order to accept this program.
- Modify the bytecode program (without removing the if statement) in order to obtain a program that is validated by the original type checker, and whose runtime behavior is exactly the same as that of the original program.

Now consider replacing the `invokevirtual A.foo()` statement with an `invoke foo()` statement that invokes method `foo()` on the class of the reference that is at the top of the operand stack.

- The verifier will fail to statically enforce that method `foo()` is defined on the type the reference at the top of the operand stack. Why?
- What happens instead if we do not have a static verifier but only dynamic checks?
- Propose a modification of the verifier in order to accept this program.
- Can you imagine why the Java bytecode designers decided to have the `invokevirtual` statement instead of something like the `invoke` statement we considered above?

3) Consider the following 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(); }}
```

- What type will be calculated for the variable `iface` of the method `xxx` during the bytecode verification?
- When can we decide that `iface.m()` is safe to call? During bytecode verification, or execution?
- What if `IFace` was a class instead of an interface? What if it was an abstract class?

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

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

- Provide a bytecode program that is rejected because of this limitation but that does not cause runtime errors.
- 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.
- How serious is this restriction from a pragmatic perspective?