Exercise 6

Traits and Bytecode verification

November 4th

1) 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) }
}
```

- 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
  ```

- We use the following code to implement a cell that stores the argument of the set method multiplied by four:
  `val e = new Cell with Doubling with Doubling`
  Why doesn't it work? What does it do? How can we make it work?
- **(Harder)** Find a modularity problem in the above, or a similar, situation. Hint: a client that gets given a class `C` does not necessarily know if a trait `T` has been mixed in that class.

2) Assume all the definitions of the previous exercise. Assume that `Cell` has the invariant that `x` is always even. Furthermore, consider a Scala method

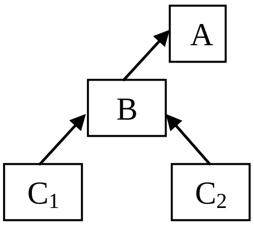
   `foo(x: Cell with Doubling with Incrementing) {…}`

- During the execution of `foo`, if we assume that all subclasses of `Cell` respect behavioural subtyping, then are we allowed to conclude that `x.get()` always returns an even number?
- 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

3) 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.

4) 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?

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?