Exercise 6

Byte Code Verification

1)

- Here ([], [E,b,b,C1,C2,A]) is initial state. We denote the type boolean as b for convenience (in reality the Java bytecode verifier views it as an integer).

0	iload_1	([b], [E,b,b,C1,C2,A])	[b], [E,b,b,B,A,A])	([b], [E,b,b,A,A,A])
1	ifeq 22	([], [E,b,b,C1,C2,A])	([], [E,b,b,B,A,A])	([], [E,b,b,A,A,A])
4	iload_2	([b], [E,b,b,C1,C2,A])	([b], [E,b,b,B,A,A])	([b], [E,b,b,A,A,A])
5	ifeq 12	([], [E,b,b,C1,C2,A])	([], [E,b,b,B,A,A])	([], [E,b,b,A,A,A])
8	aload_3	([C1], [E,b,b,C1,C2,A])	([B], [E,b,b,B,A,A])	([A], [E,b,b,A,A,A])
9	goto 14	([C1], [E,b,b,C1,C2,A])	([B], [E,b,b,B,A,A])	([A], [E,b,b,A,A,A])
12	aload 4	([C2], [E,b,b,C1,C2,A])	([A], [E,b,b,B,A,A])	([A], [E,b,b,A,A,A])
14	astore_3	([B], [E,b,b,C1,C2,A]) → ([], [E,b,b,B,C2,A])	([A], [E,b,b,B,A,A]) → ([], [E,b,b,A,A,A])	([A], [E,b,b,A,A,A]) →([], [E,b,b,A,A,A])
15	aload 5	([A], [E,b,b,B,C2,A])	([A], [E,b,b,A,A,A])	
17	astore 4	([], [E,b,b,B,A,A])	([], [E,b,b,A,A,A])	
19	goto 0	([], [E,b,b,B,A,A])	([], [E,b,b,A,A,A])	
22	aload_3	([A], [E,b,b,A,A,A])		
23	areturn	([A], [E,b,b,A,A,A])		

In the provided table, each cell contains the output value of a corresponding instruction. Different columns correspond to different iterations. There are two values for the instruction at address 14. The first one is the output of the join operation, and the second one is the output of the corresponding instruction.

- Here the essential information is marked with bold font:

0	iload_1	**([],[E,b,b,A,A,A])** →([b], E,b,b,A,A,A])
1	ifeq 22	([], [E,b,b,A,A,A])
4	iload_2	([b], [E,b,b,A,A,A])
5	ifeq 12	([], [E,b,b,A,A,A])
8	aload_3	([A], [E,b,b,A,A,A])
9	goto 14	([A], [E,b,b,A,A,A])
12	aload 4	([A], [E,b,b,A,A,A])
14	astore_3	**([A], [E,b,b,A,A,A])** →([], [E,b,b,A,A,A])
15	aload 5	([A], [E,b,b,A,A,A])
17	astore 4	([], [E,b,b,A,A,A])
19	goto 0	([], [E,b,b,A,A,A])
22	aload_3	([A], [E,b,b,A,A,A])

23	areturn	([A], [E,b,b,A,A,A])

2) Here ([], [E,T]) is the initial state, where T is an unitialized register.

0	iconst_5	([int], [E,T])
1	istore_1	([int], [E,int])
2	aload_0	([E], [E,int])
3	astore_1	([], [E,E])
4	iload_1	ERROR
5	iconst_1	
6	iadd	
7	istore_1	
8	return	

The error happens because iload_1 expects that the local variable has integer type, but its type is E.

3)

- Because the inference algorithm doesn't take interfaces into consideration, the calculated type for the variable `iface` is Object.
- Because the inferred type of the `iface` is Object the decision can be made only during the execution.
- In both cases the inferred type of the `iface` is `IFace`. The decision about the safety of the call can be made during bytecode verification.

4) Here is an example of such a program:
x=true; x=5;
The type of the variable can change in the bytecode but not in the source code.

5) No it can't be done. Here is an example in pseudo-code that demonstrates it:

```
int i = 0;
while(f(i)){
        i++;
        aload_0;
}
```

To improve readability we present the example in a mixture of Java source code and bytecode. The instruction aload_0 (pushing the value of `this` on to the top of the stack) is a bytecode instruction, and the rest is Java source code.

Here f is an arbitrary function from int to boolean. The example puts *this* in to the stack until we reach a value of i such that f(i) is false. So we need to find whether ∃ i: int | ¬f(i) is true. And this question is essentially equivalent to the halting problem. That is why it is impossible to construct an algorithm for inferring the maximal stack size.

6)

- if(b) aload_0; is a simple example. There are two possibilities for the stack size after executing the statement.
- Yes we can construct such an algorithm. The update is as follows: when joining stacks of different sizes, pick the smallest one.
- This limitation is not essential. If we have two states {[head1, x], [head2]} where head1 and head2 are stacks of the same size, then we can't access x.