

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.

— solution —

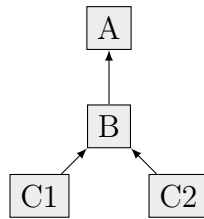
In the following, we try to verify the bytecode. `T` is an uninitialized register. A state is represented by a pair (S, R) where S describes the contents of the state and R describes the contents of registers.

```
// ([],[E,T]) -- initial state
iconst 5
// ([int],[E,T])
istore 1
// ([], [E,int])
aload 0
// ([E], [E,int])
astore 1
// ([], [E,E])
iload 1
// ERROR!
...
```

The error happens because `iload 1` expects that the local variable has integer type, but its type is `E`.

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.

— solution —

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

We show the solution following the convention of Lecture 4, Slide 21. To each command we dedicate an input and an output column. A command may have multiple inputs and outputs, corresponding to the different iterations of the algorithm. You may also want to see an animated solution of this task, published separately.

		IN	OUT
0	iload 1	([], [E,b,b,C1,C2,A]) ([], [E,b,b,B,A,A]) ([], [E,b,b,A,A,A])	([b], [E,b,b,C1,C2,A]) ([b], [E,b,b,B,A,A]) ([b], [E,b,b,A,A,A])
1	ifeq 22	([b], [E,b,b,C1,C2,A]) ([b], [E,b,b,B,A,A]) ([b], [E,b,b,A,A,A])	([], [E,b,b,C1,C2,A]) ([], [E,b,b,B,A,A]) ([], [E,b,b,A,A,A])
4	iload 2	([], [E,b,b,C1,C2,A]) ([], [E,b,b,B,A,A]) ([], [E,b,b,A,A,A])	([b], [E,b,b,C1,C2,A]) ([b], [E,b,b,B,A,A]) ([b], [E,b,b,A,A,A])
5	ifeq 12	([b], [E,b,b,C1,C2,A]) ([b], [E,b,b,B,A,A]) ([b], [E,b,b,A,A,A])	([], [E,b,b,C1,C2,A]) ([], [E,b,b,B,A,A]) ([], [E,b,b,A,A,A])
8	aload 3	([], [E,b,b,C1,C2,A]) ([], [E,b,b,B,A,A]) ([], [E,b,b,A,A,A])	([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])	([C1], [E,b,b,C1,C2,A]) ([B], [E,b,b,B,A,A]) ([A], [E,b,b,A,A,A])
12	aload 4	([], [E,b,b,C1,C2,A]) ([], [E,b,b,B,A,A]) ([], [E,b,b,A,A,A])	([C2], [E,b,b,C1,C2,A]) ([A], [E,b,b,B,A,A]) ([A], [E,b,b,A,A,A])
14	astore 3	([C1], [E,b,b,C1,C2,A]) ([B], [E,b,b,C1,C2,A]) ([B], [E,b,b,B,A,A]) ([A], [E,b,b,B,A,A]) ([A], [E,b,b,A,A,A])	- ([], [E,b,b,B,C2,A]) - ([], [E,b,b,A,A,A]) ([], [E,b,b,A,A,A])
15	aload 5	([], [E,b,b,B,C2,A]) ([], [E,b,b,A,A,A])	([A], [E,b,b,B,C2,A]) ([A], [E,b,b,A,A,A])
17	astore 4	([A], [E,b,b,B,C2,A]) ([A], [E,b,b,A,A,A])	([], [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])	([], [E,b,b,B,A,A]) ([], [E,b,b,A,A,A])
22	aload 3	([], [E,b,b,C1,C2,A]) ([], [E,b,b,B,A,A]) ([], [E,b,b,A,A,A])	- - ([A], [E,b,b,A,A,A])
23	areturn	([C1], [E,b,b,C1,C2,A]) ([B], [E,b,b,B,A,A]) ([A], [E,b,b,A,A,A])	- - ([], [E,b,b,A,A,A])

B) Provide the minimal type information that enables verification of the bytecode without a fixpoint computation.

— solution —

In the following code, we mark the types that are given by the user, and those inferred by the type checker.

```
// given: ([], [E,b,b,A,A,A])
0: iload 1
// ([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

    // ([], [E,b,b,A,A,A])
12: aload 4
    // given: ([A], [E,b,b,A,A,A])
14: astore 3
    // ([], [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
    // ([], [E,b,b,A,A,A])

```

The requirement to have type information at all basic blocks is a simplification that makes it easier to determine where the compiler should output the information. Note that some basic blocks have only a single preceding instruction, but determining this statically could be hard. Such basic blocks do not in theory need type information. Only basic blocks that are also join points definitely need type information. In our example, instructions 4, 8, 12 and 22 are indeed the beginnings of basic blocks, but there is exactly one path to enter these blocks and therefore type information is not really needed since this information will be identical to the *out*-state of the single preceding instruction.

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?

— solution —

Because the inference algorithm doesn't take interfaces into consideration, the calculated type for the variable `iface` is `Object`.

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

— solution —

Because the inferred type of the `iface` is `Object` the decision can be made only during the execution.

C) What if `IFace` was a class instead of an interface? What if it was an abstract class?

— solution —

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.

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.

— solution —

```
0 : aload 0
1 : iconst 1
2 : ifne 4
3 : aload 0
4 : astore 1
```

Note: `ifne` jumps to the given index if the integer value at the top of the stack is not equal to zero. It pops the value at the top of the stack.

There are two possibilities for the stack size after executing this program. In any of the two cases, the height of the stack at point 4 is at least 1, and there will be surely a reference value at the top of the stack.

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.

— solution —

We distinguish between two different cases.

1. If the stack sizes are statically known we can construct such an algorithm. The update is as follows: when joining stacks of different sizes, pick the smallest one, but carry as extra information the size of the largest one to be used when checking for overflow. Note that if we just picked the smaller one and used that, we would not prevent stack overflows at runtime.
If we just picked the largest one and made the “extra” values into dummy values by giving them the “top” type, we might not prevent underflows when using instructions such as `pop()`.
2. In general it is not possible to implement an algorithm that can deal with stack sizes which could vary at run-time. For example if we push elements on top of the stack in a loop, then the verifier will have no way of deciding what an upper bound for the size is. Conversely for loops which pop elements from the stack the verifier won't be able to deduce a lower bound for the stack size. These situations can easily result in over/underflows and should be rejected.

C) How serious is this restriction from a pragmatic perspective?

— solution —

This limitation is not essential. If there are two states $\{[head1, x], [head2]\}$ where `head1` and `head2` are stacks of the same size, then any following code cannot access `x` and it would have been possible to remove `x` already during bytecode generation. This is indeed what the Java compiler does. Consider the following Java code:

```
public int bar() { return 42; }
public int foo(int x) {
    if (x==0) bar();
    return x;
}
```

If `bar` is called then it will put 42 on the stack, but this value is not actually needed for the final `return` instruction. The Java compiler would emit as many `pop` instructions as necessary to remove unneeded stack elements and make sure that all paths that reach `return` have the same stack length. Here is the bytecode that corresponds to the `foo` method:

```
0: iload_1
1: ifne          9
4: aload_0
5: invokevirtual bar // Call to bar(), puts an int on the stack
8: pop           // Pop the stack to remove the unnecessary int
9: iload_1       // Here we get equal stack sizes from both paths
10: ireturn
```

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.

— solution —

```
public class List {
    Object[] elements;
    public void add(int i, Object el) {elements[i]=el;}
    public Object get(int i) {return elements[i];}
}
```

Advantages: short implementation.

Limits: the type of the method result of get is Object. When using such a class, usually we have to dynamically cast the values returned by this method.

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

— solution —

```
public interface List {
    public void add(int i, Object el);
    public Object get(int i);
}

public class IntList implements List {
    Integer[] elements;

    public void add(int i, Object el) {elements[i]=(Integer) el;}

    public Integer get(int i) {return elements[i];}
}
```

Advantages: method get returns an Integer, thus we do not need dynamic casting of the values returned by this method.

Limits: in Java, we have the same limits like before, and in addition code duplication and additional type castings and checks in method add. Moreover, we do not have behavioural subtyping, since method add in IntList may not respect the expected contracts in List. In particular, if we invoke it passing an object that is not an instance of Integer, the runtime environment would raise an exception and the element would not be added to our list.

C) Implement the list using generic types.

— solution —

```
public class List<T> {
    T[] elements;
    public void add(int i, T el) {elements[i]=el;}
    public T get(int i) {return elements[i];}
}
```

Advantages: short implementation, statically type safe.

Limits: nothing! :) we have only advantages...

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.

— solution —

```
Logger<Object> l = new Logger<Object>();
l.log(new Object());
```

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.

— solution —

```
interface Loggable {
    String loggerString();
}

class Logger<T extends Loggable> { ... }
```

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` 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?

— solution —

Yes, the code is safe.

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

- Will the method `foo` compile now?

— solution —

Yes.

- 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?

— solution —

- When calling methods of generic classes, it would be necessary to check whether the dynamic type of the result is a subtype of the static type of the variable where the result is stored.

- When reading fields of generic classes, it would be necessary to check whether the dynamic type of the field is a subtype of the static type of the variable where the object is stored.