ETH
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.

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 content of the stack and `R` describes the content of the 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 the type integer, but its type is `E`.

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)?

solution

We assume that the maximal stack size is 3 and that $MR = 3$ (since we have three parameters/local variables): **this**, one argument (`x`), and one local variable (`y`). The initial state is `([], [C, A, T])`, where `C` is the type of **this**, `A` is the type of the argument `x` and the local variable `y` is uninitialized.

```
// ([],[C,A,T])
0: iconst 7
// ([int],[C,A,T])
1: istore 2
// ([],[C,A,int])
2: aload 0
// ([C],[C,A,int])
3: aload 2
// ERROR!
...
```

The error happens because `aload 2` expects that the local variable (from register 2) has a reference type, but its type is `int`.

Let's now assume that we correct the given bytecode, such that in line 3 we have `iload 2`. All the other instructions remain unchanged. We then obtain:

```
... // as before
3: iload 2
// ([int,C],[C,A,int])
4: aload 1
// ([A,int,C],[C,A,int])
5: invokevirtual C.bar.B(int,A)
// ([B],[C,A,int])
```

So the bytecode successfully verifies.

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) **CORRECT:** `([],[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

solution

Running the bytecode type inference algorithm once from instruction 0 to instruction 2 results in retrieving the object in the second register and storing it in the third register. This object is of type `A`, so the result propagated to instruction 0 after the jump is `([],[A,A,A])`. We now need to join this state with the initial state `([],[A,A,B])`, by computing the pointwise smallest common supertype (`scs`). Since `B` is a subclass of `A`, `(scs(A, B)) = A`. Therefore the resulting input to the next iteration of the algorithm is `([],[A,A,A])`. This is then propagated to the jump instruction, reaching the fixed point. (The inference algorithm runs twice through instructions 0 and 1, and once through instruction 2, before reaching the fixed point.)

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?

solution

The inference algorithm does not take interfaces into consideration, so 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 during execution?

solution

As the inferred type of the `iface` is `Object`, the decision can be made only 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?

solution

In both cases the inferred type of the `iface` would be `IFace`. The decision about the safety of the call could be made during bytecode verification.

Task 5

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

solution

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.

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.

solution

```
public class List {
   Object[] elements = new Object[100];
   public void add(int i, Object el) {elements[i] = el;}
   public Object get(int i) {return elements[i];}
}
```

Advantages: short implementation.

Limitations: the return type of the method `get` is `Object`; when using it, we usually have to dynamically cast its return values.

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 = new Integer[100];

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

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

Advantages: the method `get` returns an `Integer`, thus we do not need dynamic casting of its return values.

Limitations: we have the same limitations as before (if programming against the interface), and in addition code duplication and further type casts/checks in the implementation of concrete list classes, e.g. in `add`. Moreover, we do not have behavioural subtyping, since the method `IntList.add` may not respect the expected contracts of `List` (due to the additional cast). For example, if we invoked `add` 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 = (T[]) new Object[100];
   public void add(int i, T el) {elements[i] = el;}
   public T get(int i) {return elements[i];}
}
```

Advantages: short implementation, statically type safe.

Limitations: none, we have only advantages :)

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.

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<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?

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.