Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Exercise 7

Bytecode Verification

November 8, 2013

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

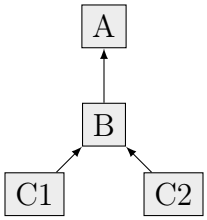
In the following, we try to verify the bytecode. `T` is an uninitialized register. A state is represented by a pair `(S,R)` where `S` describes the contents of the state and `R` describes the contents of registers.

```
  // ([],[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
```

- 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.

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.

		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,B,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])

- In the following code, we mark the types that are given by the user, and those infered 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
```

```
      // ([A], [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])
```

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();
   }
}
```

- 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?

Solution

- 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.

Task 4

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?

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.

- Yes 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()`.

  Note that in general it's 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. There situations can easily result in over/underflows and should be rejected.

- 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`.
  ```