Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Exercise 8

Bytecode Verification and Parametric Polymorphism

November 19, 2021

Task 1

(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 2

Consider the following Java code:

```
interface IFace { void m(); }

class Cl1 implements IFace {
```

```java
    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 3

(from a previous exam)

Consider an *incorrect* bytecode verifier called *BuggyVerifier*, in which due to a bug the `aload` rule assumes that the loaded element is stored at the bottom of the stack instead of at the top (see the formal description below), while all the other rules are implemented correctly.

$$\begin{array}{l} \text{aload } n : \\ \quad (S, R) \rightarrow (\underline{S.R(n)}, R), \\ \quad \text{if } 0 \leq n < MR \ \wedge \ R(n) <: Object \ \wedge \ |S| < MS \end{array}$$

Assume that the initial state (stack and registers) is $([\,], [A, B])$, with the maximum number of registers $MR = 2$ and the maximum stack size $MS = 2$. A and B are classes such that $B <: A$.

A) Write a short bytecode program that is accepted by *BuggyVerifier*, but is **not** accepted by a correct bytecode verifier. Clearly mark the line at which the correct verifier detects an error, and briefly describe the error.

solution

```
0: iconst 42
1: aload 0
2: istore 0 // ERROR: the head of the stack is not an integer
```

You can use in your solution all the bytecode operations seen during the lectures. As a reminder, here are some of them:

- `iconst` n: create on the stack a value n of type `int`.
- `iload` n: load on top of the stack an element of type `int` from the n-th register.
- `astore` n: remove an object from the top of the stack and store it in the n-th register.
- `goto` n: continue the execution from the operation at label n.

B) Is it possible that *BuggyVerifier* incorrectly accepts a program that overflows the stack, by pushing more than *MS* elements? Write yes or no, then motivate your answer.

solution

No. The rule still correctly checks that the stack size is less than MS.

Task 4

A) Compare dynamic type checking with the `dynamic` keyword to static type inference with `var` in C#:

- Give a correct program which can be realized with `dynamic` but not with `var`.

solution

```
static void Main() {
    dynamic x;
    if(condition()) {
        x = 5;
    } else {
        x = "hello";
    }

    Print(x);
}

static void Print(string str) {
    Console.WriteLine(str);
}

static void Print(int value) {
    Console.WriteLine(value);
}
```

- Give an incorrect program which will be accepted by the compiler with `dynamic` but not with `var`.

solution

```
var x = 3;
x.substring(...);
```

B) C#'s most general type is `object`. Similar to `var` and `dynamic`, you can write `object x = ...` with an expression of any type on the right-hand side.

- Given a compiling program using `var`. Can we replace all `var` keywords by `object` and add explicit casts in the right places so that the program compiles and runs as before?

solution

This will be possible in all cases where we know what the type of the variable declared with `var` is. In those cases we can just cast the declared variable in all places where it is used to the most general type fulfilling all static type constraints on the corresponding variable. Since the original program compiled, such a type must exist.

In the case of anonymous types however, we do not know the name of the type to cast to. Consider:

```
var x = new { a = 108, b = "Hello" };
Console.WriteLine(x.b);
```

Here, we could change `var` to `object`, but we will not be able to cast `x` in the second line, because we do not know the type name which the compiler generates for this anonymous type.

- Given a compiling program using `dynamic`. Can we replace all `dynamic` keywords by `object` and add explicit casts in the right places so that the program compiles and runs as before?

solution

Generally we cannot do this, as shown in the following example:

```
static void Main() {
    dynamic x;
    if(condition()) {
        x = 5;
    } else {
        x = "hello";
    }

    Print(x);
}

static void Print(string str) {
    Console.WriteLine(str);
}

static void Print(int value) {
    Console.WriteLine(value);
}
```

To make this code work with `object`, we would need to add explicit type checks and cast the argument to the proper static type.

For both questions, either informally describe how to do the replacement, or give a counterexample where the transformation will always produce a program that does not compile or behaves differently. Note that explicit casts to `dynamic` are not allowed in the transformation.

C) Assume now a language like C#, but with covariant return types and contravariant parameter types. Given four classes `A`, `B`, `C` and `D`:

```
class A { int m (int x); }
class B { void m (dynamic x); }
class C { dynamic m (int x); }
class D { dynamic m (dynamic x); }
```

Develop a subtyping rule for the `dynamic` type annotation and informally explain the reasoning behind it. What are the potential subtypes among the four classes above?

> solution
>
> Following the Substitution principle, `dynamic` is equivalent to `object`, in that it accepts any type. Therefore, the usual subtyping rules apply, treating `dynamic` as the most general supertype of all other types. The potential subtyping relations are `A <: C` and `D <: C`.
>
> There are two different ways of looking at class `B`. On the one hand, we could just say that `void` is a special keyword that indicates the absence of a return value, and thus the method `B.m` is unrelated to the other methods. Alternatively, we can allow methods with `void` return type to be overwritten by methods with any return type (assuming the parameter variance rules are satisfied): if a client code is written to expect `void` (no return value), then we could instead use a method which returns an arbitrary value and just discard it. In this second interpretation we will additionally have `D <: B`.

Task 5

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 6

(from a previous midterm)

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

```java
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:

```java
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.

Task 7

(from a previous exam)

A) Recall the Java interface `Comparable<T>` that was shown in the lecture:

```java
public interface Comparable<T> {
    public int compareTo(T other);
}
```

The method compareTo returns $\begin{cases} 1 & \text{if this is greater than other} \\ 0 & \text{if this is equal to other} \\ -1 & \text{if this is less than other} \end{cases}$

Suppose we want to turn Comparable into an abstract class with an additional helper method greaterThan, that returns true if and only if this is greater than other.

Assume the following implementation:

```java
public abstract class Comparable<T> {
    public abstract int compareTo(T other);

    public boolean greaterThan(T other) {
        return other.compareTo(this) < 0;
    }
}
```

A.1) Why does this implementation not type check?

solution

T requires an upper bound that provides compareTo.

A.2) Fix the type error by changing only the body of greaterThan, while preserving the intended semantics of the method.

solution

```java
return this.compareTo(other) > 0;
```

A.3) Fix the type error by changing only the class signature and the signature of the method compareTo.

solution

Let the class signature be Comparable<T extends Comparable<T>> and the signature of compareTo be compareTo(Comparable<T> other).

B) Suppose we have the following class:

```java
class A<X                    ,Y                        > {
    X a;
    Y b;
}
```

Consider a variable v whose type is A<S,T> where S and T satisfy the type bounds that you have to insert above. Your type bounds have to guarantee that for all sequences of a and b accesses on v (e.g., v.b.a.b.a.a.b.b) the following two properties hold:

- The static type of a sequence ending in a is S.
- The static type of a sequence ending in b is T.

solution

```java
class A<X extends A<X,Y>,Y extends A<X,Y>> {
    X a;
    Y b;
}
```