Concepts of Object-Oriented Programming
AS 2022

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Exercise 3
Subtyping
October 14, 2022

Task 1

Consider the following types written in Java-like syntax:

```
class D { A a; B foo(A a) throws RuntimeException; }

class F { C a; A foo(B a) throws RuntimeException; }

class G { A a; A foo(C a) throws Exception; }

class H { A a; A foo(A a) throws IndexOutOfBoundsException; }
```

Assume that `B <: A`, `C <: A`, and `IndexOutOfBoundsException <: RuntimeException <: Exception`. Which of the following subtype relations could be allowed by a sound type system?

(a) `D <: H` and `H <: G`

(b) `D <: G` and `H <: F`

(c) **CORRECT:** `D <: G` and `H <: G`

(d) `H <: D` and `H <: F`

(e) None of the above

Task 2

Assume the following class definitions in a nominally typed language:

```
class A {...}
class B extends A {...}
```

Consider now the following two classes:

```
class Super {
  B foo(B x) { return x; }
}

class Sub extends Super {
  A foo(A x) { return x; }
}
```

According to the rules presented in the lecture, this subtyping is illegal. Briefly explain why this is the case. However, considering the substitution principle, this subtyping is *safe*. Why?

solution

According to the rules from the lecture, overriding methods should have covariant results. The method `Sub.foo()` returns an `A` and $\texttt{A} \not<: \texttt{B}$, so this class declaration should be rejected by the type checker.

However, (1) `Sub` has a wider interface then `Super`, since any client code that uses an instance of `Super` can pass as parameter to `foo` only an object of type `B`, and (2) at runtime `Sub.foo()` returns the same object that `Super.foo()` would return. This means that the substitution of an instance of `Super` with an instance of `Sub` would not be noticeable at runtime and would cause no errors. So, the substitution principle holds. Note that this subtyping is very fragile. Any modification to the implementation of `Super.foo()` or `Sub.foo()` might break the substitution principle, even if the signature of the methods remains the same.

Task 3

In this question, we are in a nominal subtyping setting. Some languages have a special type `MyType` that represents the *dynamic type* of `this` object.

(a) Consider the following code:

```
class Point {
    int x,y;
    boolean equals(MyType other) {
        return x == other.x && y == other.y;
    }
}

class ColorPoint extends Point {
    int color;
    override boolean equals(MyType other) {
        return super.equals(other) && color == other.color;
    }
}
```

This definition requires that the dynamic type of the parameter of `equals` is a subtype of the dynamic type of `this`.

Consider the following definitions that give static types to some variables:

```
Point p;
ColorPoint cp1, cp2;
```

and the following calls:

```
p.equals(cp1)    // A
p.equals(cp2)    // B
cp1.equals(p)    // C
cp2.equals(cp1)  // D
cp1.equals(cp2)  // E
```

Assume a sound, statically-checked type system. Which of the calls above must be forbidden and which may be allowed? Why?

solution

All calls are potentially unsafe and should be forbidden. The reason is that the dynamic type of both the receiver and the parameter are unknown and are not guaranteed to match the restriction that the dynamic type of the parameter should be a subtype of the dynamic type of the receiver.

(b) Answer the same question, assuming that `ColorPoint` is *final*, i.e., we may not declare new classes as its subtypes.

solution

In this case, we know that the dynamic types of both `cp1` and `cp2` are `ColorPoint`. This guarantees that the calls D and E are ok. However, the first three calls remain unsafe. The first two calls are unsafe because the dynamic type of `p` may be a subtype of `Point` that has no relation to `ColorPoint`. The C call is not safe because `p` may have the dynamic type `Point`.

(c) Assume now that the language includes the feature of *exact types*. An exact type is written `@C` where `C` is a normal type. When we declare that an object `o` is of type `@C`, then `o` is of type `C`, but does not belong to any of the other subtypes of `C`. Assume that the definitions of our variables are changed as follows:

```
@Point p;
@ColorPoint cp1;
ColorPoint cp2;
```

Do not assume that `ColorPoint` is final. Which calls should be forbidden now? Why?

solution

All is known about the dynamic types of `cp1` and `p`. The calls A, B, and E are safe. D is not, because `cp2` may belong to a proper subtype of `ColorPoint`. C is not, because `p` is of dynamic type `Point`.

Hint. The classes shown here may be subclassed in code that is not available. The type-checker *cannot* make the assumption that there are no other class definitions elsewhere.

Task 4

Assume we have a Java-like language with contravariant parameters where methods can declare *default values* for some of their parameters as follows:

```
class Person {
    public static Person create(
        String firstName,
        String lastName = "None",
        int age = -1
    ) { ... }
}
```

After the first parameter that has a default value, all subsequent parameters must also have a default value. If a method call has i arguments and the called method has j parameters, where $i \leq j$, the default values are used for the parameters $i+1, \ldots, j$. A call `Person.create("Emma")` is therefore equivalent to the call `Person.create("Emma", "None", -1)`.

A call with fewer arguments than parameters without default values leads to an error, and so does a call with more arguments than there are parameters. Note that a method with default arguments is a single method, **not** a set of methods with different numbers of parameters. You can assume that `String` is a final class.

A) Consider the following code. Assume `C <: B <: A`, and that there are the public constants `A_OBJ` of type `A`, `B_OBJ` of type `B`, and `C_OBJ` of type `C`.

```
class E {
    public B m(B b, A a = C_OBJ) { ... }
```

```
}

class E1 extends E {
    public C m(A b, A a) { ... }
}

class E2 extends E {
    public B m(B b = C_OBJ, A a = A_OBJ, C c = C_OBJ) { ... }
}

class E3 extends E {
    public C m(A b = B_OBJ) { ... }
}
```

A.1 Name all of the subclass declarations shown above that should be rejected by the compiler. For each of them, write a code snippet that results in a runtime type error if the code is admitted.

solution

E1: E e = new E1(); e.m(new B());

E3: E e = new E3(); e.m(new B(), new A());

A.2 What should be the general type rule for overriding in this language?

solution

Subclass method can accept any number of arguments that is accepted by the superclass, each parameter is contravariant, default values comply with the parameter type, return type is covariant.

B) Now assume the same language also has a new way of passing arguments: When using the star operator `*` on an array in an argument list, the elements of the array will be passed as single arguments. For example, if `a` is an array containing the values `v1` and `v2`, the call `m(x, y, *a)` is equivalent to `m(x, y, v1, v2)`. The expansion of arguments with star operators happens *before* default arguments are added to the call.

Consider the following class:

```
class Foo {
    public void bar(A a, B b1, C c = C_OBJ) {...}
}
```

Let `foo` be a reference to an object of type `Foo`. The constants `A_OBJ` etc. exist as before and the subtype relation is still `C <: B <: A`. Assume that the compiler accepts each of the following code snippets **B.1**, **B.2**, and **B.3**. Consider only cases where the array references are not null and the arrays do not contain null. For each of them, answer the following two questions:

1. Is it possible that the call will succeed at runtime? If so, give an example of an array that will make the call succeed.
2. Is it possible that the call will fail at runtime? If so, specify the properties that must be checked at runtime and *cannot* be checked statically to catch all possible type errors resulting from the call.

B.1

```
void baz(C[] cArray) {
   foo.bar(A_OBJ, *cArray);
}
```

solution

1. possible, `[C_OBJ]`.
2. possible, `cArray.length >= 1 && cArray.length < 3`.

B.2

```
void baz(A[] aArray) {
   foo.bar(*aArray);
}
```

solution

1. possible, `[C_OBJ, C_OBJ]`.
2. possible, `aArray.length >= 2 && aArray.length <= 3 && aArray[1] instanceof B && aArray.length > 2 ==> aArray[2] instanceof C`.

B.3

```
void baz(String[] stringArray) {
   foo.bar(A_OBJ, B_OBJ, *stringArray);
}
```

solution

1. possible, `[]`.
2. possible, `stringArray.length == 0`.

Task 5

Consider the following code written in a programming language supporting structural subtyping and the access modifiers from Java:

```
class A { private A foo(A o) {...} }

class B { protected A foo(A o) {...} }

class C { A foo(A o) {...} }

class D { protected B foo(A o) {...} }

class E { A foo(B o) {...} }
```

The `{...}` blocks represent the actual implementation of `foo`, which you can assume to be correct, but not necessary identical for all the classes. Which of the following subtype relations does not hold?

(a) `B <: A`

(b) **CORRECT:** `E <: A`

(c) `C <: E`

(d) `D <: C`

(e) More than one of the above

Now also consider the following class:

```
class F { }
```

Explain why F $\not<:$ A. Write a client code that would result in a runtime error if subtypes were allowed to remove private methods from their supertypes (that is, if F were a subtype of A). In your solution, you can add one new method with the same signature to both A and F.

solution

F is not a subtype of A because it narrows the interface of A.

We can add a method bar to both A and F and construct the client code shown below:

```
class A {
    private A foo(A o) {...}

    public boolean bar (A a) {
        return a.foo(a) == null;
    }
}

class F {
    public boolean bar (A a) {
        return true;
    }
}

A a = new A();
F f = new F();
a.bar(f); // fails because F does not have a method foo
```

Task 6

Consider the following declarations in Java:

```
interface List {
   int getSize();
}

interface Iterator {
   boolean done();
   int getCurrent();
   void next();
   void attach(List l);
}
```

List represents sequences of integers and Iterator represents a specific traversal of a list. An implementation of an iterator starts iterating over the elements of a list by first calling method attach. The following example prints all the elements found during the iteration:

```
void foo(Iterator iter, List list) {
   iter.attach(list);
   while(!iter.done()) {
      print(iter.getCurrent());
      iter.next();
   }
}
```

Does foo typecheck in Java?

Suppose that we want to have different implementations of lists. For example a linked list and an array are two different ways to implement the List interface. What problem would that

cause to the implementers of the iterators? What problem would that cause to the method `foo`?

solution

The code typechecks.

According to the given specification, an iterator must be able to work with all possible implementations of lists. This is impossible to achieve without downcasting to specific implementations, given that the interface `List` is so small. This solution is also impossible, since not all implementations of `List` are necessarily known at any given point in the program.

A possible problem for `foo` is an implementation of `Iterator` that throws an exception for unknown list implementations.