Concepts of Object-Oriented Programming
AS 2016

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Exercise 13

Self-Study Exercise Sheet

NOTE: 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 final exam. If you have any questions regarding this sheet, please consult your assistant.

Subtyping and Behavioral Subtyping

Task 1

Consider the class `X` and its only method `foo` where `ZZZ` is placeholder for a class name:

```
class X {
   /// requires x>0 ∧ (¬∃ i,j: int | 2 ≤ i,j ≤ x ∧ i*j=x)
   /// ensures result>0 ∧ result % 2 = 0
   int foo(final int x){ return (new ZZZ()).foo(x); }
}
```

Which of the four classes below could be substituted for `ZZZ` such that no contracts will be violated?

(a)
```
class A {
      /// requires x≥0
      /// ensures result = x+1
      int foo(final int x) {...}   }
```

(b)
```
class B {
      /// requires true
      /// ensures result%2 = 0
      int foo(final int x) {...}   }
```

(c)
```
class C {
      /// requires x%2 = 1
      /// ensures result = x+1
      int foo(final int x) {...}   }
```

(d) **CORRECT:**
```
class D {
      /// requires true
      /// ensures result = x*(x+1)
      int foo(final int x) {...}   }
```

Task 2

Consider the following Java classes and interfaces:

```
public interface IA { IA g(IA x); }

public interface IB extends IA { IB g(IA x); IA g(IB x); }

public interface IC extends IA { IC g(IB x); }

class B implements IB
{
   public IB g(IA x){System.out.print("B1");return null;}
   public IC g(IB x){System.out.print("B2");return null;}
}

class C implements IC
{
   public IC g(IA x){System.out.print("C1");return null;}
   public  C g(IB x){System.out.print("C2");return null;}
}


class Main{
   public static void main(String[] args) {
      B  b  = new B();
      C  c  = new C();
      IA a1 = b;
      IA a2 = c;

      IA r1 = a1.g(a2);
      IA r2 = a2.g(b);
      IC r3 = b.g(b);
      IA r4 = c.g(a2);
      C  r5 = c.g(b);
   }
}
```

What is the output of the execution of the `Main.main` method? Explain your answer.

solution

The code will print `B1 C1 B2 C1 C2`:

`a1` is of static type `IA` and dynamic type `B`, `a2` is of static type `IA`: `a1.g(a2)` maps to `IA.g(IA)`, which is overriden in `IB` as `IB.g(IA)` and then in `B` as `B.g(IA)`.

`a2` is of static type `IA` and dynamic type `C`, `b` is of static type `B`: `a2.g(b)` maps to `IA.g(IA)`, which is overriden in `IC` as `IC.g(IA)` and then in `C` as `C.g(IA)`.

`b` is of static type `B` and dynamic type `B`: `b.g(b)` maps to `B.g(IB)` (more specific than `B.g(IA)` - overload resolution).

`c` is of static type `C` and dynamic type `C`, `a2` is of static type `IA`: `c.g(a2)` maps to `C.g(IA)`.

`c` is of static type `C` and dynamic type `C`, `b` is of static type `B`: `c.g(b)` maps to `C.g(IB)` (more specific - overload resolution).

Task 3

Consider the following generic Java class:

```
class Generic<T>
{
   <U> void m1(List<? extends T> a, List<? super U>   b, U c) {...};
   <U> void m2(List<? super T>   a, List<? extends U> b, U c) {...};
   <U> void m3(List<? super U>   a, List<U>           b) {...};
}
```

Which of the method signatures of this class can be rewritten by using only generic method arguments instead of wildcards?

(a) Only `m1`

(b) Only `m2`

(c) Only `m1` and `m2`

(d) **CORRECT:** Only `m1` and `m3`

(e) Only `m2` and `m3`

(f) None of them

Task 4

Consider the following Java definitions:

```
class A{}
class B extends A{}
class L<F extends A>{}
```

And the following code that uses these definitions:

```
   void f(L<? extends A> l1, L<? extends A> l2){}

   <T extends A> void g(L<T> l1, L<T> l2){}
```

Which method can accept more input types than the other?

(a) **CORRECT:** `f` can accept more inputs than `g`

(b) `g` can accept more inputs than `f`

(c) `f` and `g` can accept the same set of input types

(d) Each can accept some input the other cannot

Task 5

Assume two Java classes `A` and `B` and assume that `B` is a subclass of `A`. Consider the following byte code:

```
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 contents of the stack and the second list indicates the contents 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

Task 6

Suppose that we have a language with the information hiding rules of Java, but with structural subtyping. What should be the subtyping relations between the following three classes?

```
class A {int foo();}
class B {protected int foo();}
class C {public int foo();}
```

solution

The subtyping relations are as follows: `C <: B <: A`

Using structural subtyping we require that the methods and fields of subclasses are more accessible than those of superclasses. When dealing with access modifiers, this means that methods with more permissive modifiers may override methods with less permissive modifiers.

Task 7

Consider the class `Hour`, defined as follows:

```
public class Hour {
   protected int h=0;
   /// invariant h>=0 && h<24

   public void set(int h) {
      if(h>=0 && h<24) this.h=h;
   }
}
```

What is the external interface of `Hour`?

solution

The external interface is composed only of the method `public set(int)` since this is the only public element of class `Hour`.

Can we extend the code, without changing the class, so that the invariant is broken? If yes, provide an example, and propose how to fix the class.

solution

The invariant can be broken easily by extending class `Hour`, and accessing the field `h` directly. For instance:

```
public WrongHour extends Hour {
   public WrongHour() {super.h=-1;}
}
```

This can be prevented by making the field `h` private.

Task 8

Consider the following class definitions in the context of the read-only type system taught in the course:

```
class C {
   public D f;
   void foo(readonly C other) {...}
}

class D { E g; }

class E {}
```

Let `a` and `b` be non-null references of type `C`. Which of the following statements is true:

(a) The call `a.foo(b)` is guaranteed not to change the value of `b.f`, but may change the value of `b.f.g`

(b) The call `a.foo(b)` is guaranteed not to change the value of `b.f` and neither the value of `b.f.g`

(c) The assignment `other.f.g = new E();` may appear in the code of `foo`

(d) **CORRECT:** None of the above is correct

Task 9

In the following question we do not consider the owners-as-modifiers discipline. We are only concerned with the topology of the ownership type system.

Consider the assignment:

```
o.f = p.g;
```

and assume that `o.f` and `p.g` have the same static type.

A) The assignment is forbidden if `o.f` has ownership modifier `lost`. Show an example to demonstrate why we need this rule to preserve the topological invariant.

solution

The following code breaks the acyclicity requirement for the topology:

```
class C
{
  rep C down;

  void foo()
  {
    down.down = this;
  }
}
```

B) If the ownership modifier of `o.f` is `any`, then what are the requirements for the assignment to be legal?

solution

None. The assignment is always legal.

C) If `o.f` has ownership modifier `lost` can we upcast `o.f` to an `any` reference and make the assignment legal? Why (not)?

solution

We cannot upcast a reference that is being assigned to. This is illegal according to the subtyping rules.

Task 10

Consider a Java class `Vector`, representing a 2 dimensional vector:

```
public class Vector {
   public Number x; // Remark: Number is a super-interface for
   public Number y; //   Integer, Double, etc.

   public Vector (Number x, Number y) {
      this.x = x;
      this.y = y;
   }
}
```

Suppose that in some other class we write the following method to calculate the length of the vector represented by a `Vector` object:

```
public double vectorLength(Vector c) {
   double x = c.x.doubleValue();
   double y = c.y.doubleValue();
   return Math.sqrt(x * x + y * y);
}
```

A) This implementation is unsafe - when executed it may throw exceptions. Why? Is this a reasonable behavior?

solution

If `c` were `null`, the field dereferences `c.x` and `c.y` would generate exceptions. Furthermore, if `c.x` were `null` then method call `c.x.doubleValue()` would generate an exception. Similarly, if `c.y` were `null`.

There is no reasonable answer for the method to return if it encounters null values - any attempt to deal with these cases would have to return some arbitrary value, since the question the method is meant to answer is undefined in these cases.

B) Add a pre-condition for the method, specifying what is required to be safe.

solution

requires: c≠**null** ∧ c.x≠**null** ∧ c.y≠**null**

C) Suppose you are allowed to modify the signature of the method to include non-null type annotations. To what extent can you weaken the necessary pre-condition?

solution

```
public double vectorLength(Vector! c)
```

would make the following pre-condition sufficient:

requires: c.x≠**null** ∧ c.y≠**null**

D) Suppose that you are also allowed to upgrade the class `Vector` to include reasonable non-null type annotations. How does this affect your previous answer? Do these changes to the class seem reasonable?

solution

By changing the types of the fields `x` and `y` to be `Number!` we could guarantee that no pre-condition would be required. This seems a reasonable change, since a `null Vector` doesn't seem to be meaningful anyway.

Task 11

Consider the following three classes (declared in the same package):

```
public class Person {
   Dog? dog; // people might have a dog

   public Person() { }
}

public class Dog {
   Person! owner; // Dogs must have an owner
   Bone! bone;    // Dogs must have a bone
   String! breed; // Dogs must have a breed

   public Dog(Person owner, String breed) {
      this.owner = owner;
      this.bone = new Bone(this);
      this.breed = breed;
   }
}

public class Bone {
   Dog! dog;        // Bones must belong to a dog..

   public Bone(Dog toOwn) {
      this.dog = toOwn;
   }
}
```

A) Annotate the code with non-null and construction type annotations where they are necessary. Explain why the code now type-checks according to construction types.

solution

Here are the annotations for the first version of the code:

```
public class Person {
   Dog? dog;   // a person might have a dog

   public Person() { }
}

public class Dog {
   Person! owner;  // A dog must have an owner
   Bone! bone;     // A dog must have a bone
   String! breed;  // A dog must have a breed

   public Dog(unc Person ! owner, unc String ! breed) {
      this.owner = owner;
      this.bone = new Bone(this);
      this.breed = breed;
   }
}
```

```
public class Bone {
   Dog! dog;         // Bones must belong to a dog..

   public Bone(unc Dog ! toOwn) {
      this.dog = toOwn;
   }
}
```

Note that we choose the parameter to the construction of `Bone` to be unclassified - since it is public then it probably should be callable with a committed parameter from client code, but it is also called inside the body of the constructor of `Dog`, with a free parameter. Note that the returned reference from these two kinds of call will be different - committed in the former case, and free in the latter. For the `Dog` constructor, we can also choose to make the parameters unclassified. Although in this case we do not directly need to permit "free" arguments being passed to the constructor, we may as well be as permissive as possible. In general, if it is possible to type a constructor body using "unclassified" argument types then this should be the preferred choice of signature as it is the most permissive. Note that the same does not apply for method signatures, since any overriding method definitions are then also be forced to cope with unclassified arguments, which may be much less convenient than using committed ones.

B) Could we provide constructors for classes `Dog` and `Bone` with no parameters?

solution

It isn't reasonable to have constructors for `Dog` and `Bone` without parameters, since we need some way of initialising their non-null fields. Although it would be possible to do this by calling e.g., the `Person` constructor from the `Dog` constructor, this doesn't seem very intuitive (nor would it be easy to establish the intuitive invariants of the code - that a `Dog`'s owner refers back to the same `Dog`, etc.). In particular, if all of the constructors need to take no parameters, they would need to call each other infinitely. This is because, we can't set up a cyclic object structure without some kind of mutual initialization (in this case we can only build an infinite object structure to satisfy the non-null requirements of all the objects).

C) Now, suppose a (possibly mad) scientist wants to extend the implementations of these classes with some genetic engineering. Firstly, we want to be able to "clone" a bone. We can add the following method to class `Bone` to make a copy of an existing bone, and assign it to another `Dog`:

```
public Bone clone(Dog toOwn) {
      return new Bone(toOwn);
   }
```

However, our scientist would like to go further, and be able to clone dogs. A cloned `Dog` should also have its bone cloned along with it, but may be assigned to a new owner: we add the following extra constructor and method to class `Dog`:

```
Dog(Dog toClone, Person newOwner) {
   this.owner = newOwner;
   this.breed = toClone.breed;
   this.bone = new Bone(this);
}

public Dog clone(Person toOwn) {
```

```
    return new Dog(this, toOwn);
}
```

However, our scientist would like to go still further, and be able to clone people. A cloned `Person` should also have its dog (if any) cloned along with it: we add the following extra constructor and method to class `Person`:

```
Person(Person toClone) {
    Dog? d = toClone.dog;
    if(d!=null) {
        this.dog = new Dog(d, this);
    }
}

public Person clone() {
    return new Person(this);
}
```

Annotate this extra code with appropriate non-null and construction types annotations. You should guarantee that each of the `clone` methods (which belong to the public interface) return a committed reference. You should ensure that your answers guarantee that all of the code type-checks - explain your choices.

Hint: think carefully about how constructor calls are typed, and what happens if the constructors are called in more than one situation.

solution

Here is the fully annotated code for the cloning case:

```
public class Person {
    Dog? dog;   // A person might have a dog

    public Person() { }

    Person(Person! toClone) {
        Dog d? = toClone.dog;
        if(d != null) {
            this.dog = new Dog(d, this);
        }
    }

    public Person! clone() {
        return new Person(this);
    }
}
public class Dog {
    Person! owner; // A dog must have an owner
    Bone! bone;    // A dog must have a bone
    String! breed; // A dog must have a breed

    public Dog(unc Person ! owner, unc String ! breed) {
        this.owner = owner;
        this.bone = new Bone(this);
        this.breed = breed;
    }

    Dog(Dog! toClone, unc Person! newOwner) {
        this.owner = newOwner;
        this.breed = toClone.breed;
        this.bone = new Bone(this);
    }
```

```
    public Dog! clone(Person! toOwn) {
        return new Dog(this, toOwn);
    }

}
public class Bone {
    Dog! dog;         // A bone must belong to a dog..

    public Bone(unc Dog ! toOwn) {
        this.dog = toOwn;
    }

    public Bone! clone(Dog! toOwn) {
        return new Bone(toOwn);
    }
}
```

Note that all parameters to the new constructors and methods need to have non-null type annotations, since they are each either dereferenced, used to initialize non-null-declared fields or passed on as further parameters to calls that require non-null parameters.

The `toClone` parameter of the new constructor of `Person` needs to be a committed parameter, otherwise when we dereference `toClone.dog` we will obtain an unclassified value, which will not be suitable to use as a parameter for the new `Dog` constructor.

The `toClone` parameter of the new constructor of `Dog` needs to be a committed parameter, since when a field is read from it, we need to obtain a result with a non-null type. However, the `newOwner` parameter of the new constructor of `Dog` needs to be an unclassified parameter. This is because this parameter is sometimes supplied from a free reference (in the new constructor of `Person`), and sometimes from a committed reference (in the clone method of `Dog`).

For similar reasons, the `toOwn` parameter of the constructor of `Bone` needs to be an unclassified parameter (as was suggested for the previous part of the question). This is because this parameter is sometimes supplied from a free reference (in the new constructor of `Dog`), and sometimes from a committed reference (in the clone method of `Bone`).

This is an important usage of the unclassified types in the construction types system - they are useful for constructors which get called sometimes with free and sometimes with committed parameters. Recall that the type of a new expression is determined from the static types of the actual parameters at a particular call, and not from the formal parameters in the constructor signature. For example, in the `clone` method of the `Bone` class, the new expression `new Bone(toOwn)` is given a committed type because the actual parameter `toOwn` has a static type which is committed, despite the fact that the constructor argument type is declared as unclassified in its signature. This means that the same constructor can produce committed/free results depending on the particular arguments provided in each call (`new` expression). In particular, the return type of the `clone` method can be a committed reference, as required in the question (the same applies to all of the clone methods in the code, since they each call constructors with only committed arguments).

Task 12

Which of the following is the *defining* characteristic of reflection?

(a) It allows for much simpler code

(b) It enables more flexibility

(c) **CORRECT:** It allows a program to observe and modify its own structure and behavior

(d) It is not statically safe

(e) It may hurt performance

(f) None of the above

Task 13

Consider the following Java code:

```
void foo() throws java.lang.Exeption {
  LinkedList<String> xs = new LinkedList<String>();
  xs.add("A"); xs.add("B"); xs.add("C");

  Class<?> c = xs.getClass();
  Method remove = c.getMethod("remove");
  xs.add(remove.invoke(xs));
}
```

which uses the following methods of class `LinkedList<E>`

```
  public E remove()
  public boolean add(E e)
```

Which of the following statements is true? The invocation of ...

(a) `c.getMethod("remove")` is rejected by the compiler

(b) `c.getMethod("remove")` raises an exception (at run time)

(c) `remove.invoke(xs)` is rejected by the compiler

(d) `remove.invoke(xs)` raises an exception (at run time)

(e) **CORRECT:** `xs.add(...)` is rejected by the compiler

(f) `xs.add(...)` raises an exception (at run time)